# THE INTEL 32-BIT MICROPROCESSORS

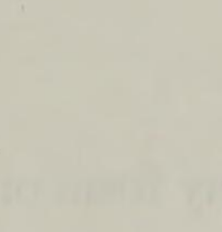
## 80386, 80486, AND PENTIUM

BARRY B. BREY

# THE INTEL 32-BIT MICROPROCESSORS

## 80386, 80486, and Pentium

BARRY B. BREY
DeVry Institute of Technology



Prentice Hall

*Englewood Cliffs, New Jersey  Columbus, Ohio*

**Library of Congress Cataloging-in-Publication Data**

Brey, Barry B.
The Intel 32-bit microprocessors : 80386, 80486, and Pentium / by Barry B. Brey.
p. cm.
Includes biographical references and index.
ISBN 0–02–314260–X
1. Intel 80386 (Microprocessor) 2. Intel 80486 (Microprocessor) 3. Pentium (Microprocessor) I. Title.
QA76.8.I2684B74 1995
004.16—dc20 94-14162
CIP

Editor: David Garza
Production Editor: Mary Irvin
Production Buyer: Deidra Schwartz
Production Coordination: Custom Editorial Productions, Inc.

This book was set in Times Roman and was printed and bound by Courier Printing/The Murray Company. The cover was printed by Phoenix Color Corp.

© 1995 by Prentice-Hall, Inc.
A Simon & Schuster Company
Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-02-314260-X

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S. A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

# PREFACE

In the memory of my Grandfather—Wayne A. Feather, an extremely learned and successful businessman—who was forced to leave school for work after completing only a fifth-grade education, yet zealously cherished knowledge and education.

# PREFACE

This text is written for the student in a course of study that requires a thorough knowledge of programming and interfacing of the Intel family of microprocessors. It is a very practical reference text for anyone interested in the programming and interfacing aspects of this important microprocessor family. Today, anyone functioning or striving to function in a field of study that uses computers must understand assembly language programming and interfacing. Intel microprocessors have gained wide applications in many areas of electronics, communications, control systems, and particularly in desktop computer systems.

## ORGANIZATION AND COVERAGE

In order to cultivate a comprehensive approach to learning, each chapter of the text begins with a set of objectives that briefly defines the contents of the chapter. This is followed by the body of the chapter, which includes many programming applications to illustrate the main topics of the chapter. At the end of each chapter a numerical summary, which doubles as a study guide, reviews the information presented in the chapter. Finally, questions and problems are provided to promote practice and mental exercise with the concepts presented in the chapter.

This text contains many example programs using the Microsoft Macro Assembler program to provide an opportunity to learn how to program the Intel family of microprocessors. Operation of the programming environment includes the linker, library, macros, DOS function, and BIOS functions.

Also provided is a thorough description of each family member, memory systems, and various I/O systems that include disk memory, ADC and DAC, USARTs, PIAs, timers, keyboard/display controllers, arithmetic coprocessors, and video display systems. Through these systems, the reader will learn a practical approach to microprocessor interfacing.

## APPROACH

Because the Intel family of microprocessors is quite diverse, this text initially concentrates on real-mode assembly language programming, which is compatible with all versions of

the Intel family of microprocessors. Instructions for each family member (the 80386, 80486 and Pentium) are compared and contrasted with the 8086/8088 microprocessor. This entire series of microprocessors is very similar, so more advanced versions can easily be learned once the basics are understood.

In addition to fully explaining the programming and operation of the microprocessor, this text also explains the programming and operation of the numeric coprocessor (8087/80287/80387/80486/7 and Pentium). The numeric coprocessor provides access to floating-point calculations that are important in applications such as control systems, video graphics, and computer aided design (CAD). The numeric coprocessor allows a program to access complex arithmetic operations that are otherwise difficult to achieve with normal microprocessor programming.

Also described are the pinouts and function of the 80386, 80486, and Pentium microprocessors. Interfacing is first developed using the 80386/80486 with some of the more common peripheral components. After learning the basics, a more advanced emphasis is placed on the Pentium microprocessor. Coverage of earlier versions of the Intel microprocessors has been minimized so the 80386, 80486, and Pentium can be covered in complete detail.

This approach—covering the operation of the microprocessor and programming with the advanced family members, along with the interfacing of all family members—provides a working and practical background on the Intel family of microprocessors. On completion of a course of study based on this text, you should be able to:

1. Develop control software to control an application interface to the 8086/8088, 80286, 80386, 80486, and Pentium microprocessor. Generally, the software developed will also function on all versions of the microprocessor. This software also includes DOS-based applications.
2. Program using DOS function calls to control the keyboard, video display system, and disk memory in assembly language.
3. Use the BIOS functions to control the keyboard, display, and various other components in the computer system.
4. Develop software that uses interrupt hooks and hot-keys to gain access to terminate and stay resident software.
5. Program the numeric coprocessor to solve complex equations.
6. Explain the differences between the family members and highlight the features of each member.
7. Describe and use real- and protected-mode operation of the 80286, 80386, 80486, and Pentium microprocessors.
8. Interface memory and I/O systems to the microprocessor.
9. Provide a detailed and comprehensive comparison of all family members and their software and hardware interface.
10. Explain the operation of disk and video systems.

## CONTENT OVERVIEW

Chapters 1 and 2 introduce the Intel family of microprocessors with an emphasis on the microprocessor-based computer system. These early chapters introduce the microprocessor,

its history, its operation, and the methods used to store data in a microprocessor-based system. They also explore the programming model of the microprocessor and system architecture. Both real- and protected-mode operation are explained in Chapter 2.

Once an understanding of the basic machine is grasped, Chapters 3 through 6 explain how each instruction functions with the Intel family of microprocessors. As instructions are explained, simple applications are presented to illustrate the operation of the instructions and develop basic programming concepts.

Once the basis for programming is developed, Chapter 7 provides applications using the assembler program. These applications include programming using DOS and BIOS function calls. Disk files are explained, as well as keyboard and video operation on a personal computer system. This chapter provides the tools required to develop virtually any program on a personal computer system. It also introduces the concept of interrupt hooks and hot-keys.

Chapter 8 details the operation and programming of the 8087–Pentium family of arithmetic coprocessors. Today, few applications function efficiently without the power of the arithmetic coprocessor.

Chapter 9 introduces the 80386 and 80486 microprocessors as a basis for learning basic memory and I/O interfacing, which follow in later chapters. This chapter shows the buffered system as well as the system timing.

Chapter 10 provides complete detail on memory interface, using both integrated decoders and programmable logic devices. Parity is illustrated, as are dynamic memory systems. The 8-, 16-, 32-, and 64-bit memory systems are provided so the 8086–Pentium microprocessors can be interfaced to memory. Also discussed are interleaved memory systems and cache systems.

Chapter 11 provides a detailed look at basic I/O interfacing that includes PIAs, timers, keyboard/display interfaces, USARTs, and ADC/DAC. It also describes the interface of both DC and stepper motors.

Once these basic I/O components and their interface to the microprocessor is understood, Chapters 12 and 13 provide detail on advanced I/O techniques that include interrupts and direct memory access (DMA). Applications include a printer interface, real-time clock, disk memory, and video systems.

Chapter 14 provides detail on the advanced Pentium microprocessor. In this chapter we explore the differences between the 80386/80486 and the Pentium. Also described are memory management and memory paging.

Appendices are included to enhance the application of the text. These include:

A. A complete listing of the DOS INT 21H function calls. This appendix also details the use of the assembler program and many of the BIOS function calls, including BIOS function call INT 10H.
B. Complete listing of all 8086/8088/80286/80386/80486/Pentium instructions including many example instructions and machine coding in hexadecimal, as well as clock timing information.
C. A compact list of all the instructions that change the flag bits.
D. Answers for the even-numbered questions and problems.

# CONTENTS


1 INTRODUCTION TO THE MICROPROCESSOR AND COMPUTER 1

1–1 A Historical Background 2
1–2 The Microprocessor-Based Computer System 10
1–3 Computer Data Formats 23
1–4 Summary 33
1–5 Questions and Problems 35

2 THE MICROPROCESSOR AND ITS ARCHITECTURE 38

2–1 80386, 80486, and Pentium Internal Architectures 39
2–2 Real-Mode Memory Addressing 44
2–3 Protected-Mode Memory Addressing 50
2–4 Summary 55
2–5 Questions and Problems 56

3 ADDRESSING MODES 59

3–1 Data-Addressing Modes 60
3–2 Program Memory-Addressing Modes 81
3–3 Stack Memory Addressing 83
3–4 Summary 86
3–5 Questions and Problems 88

4 DATA MOVEMENT INSTRUCTIONS 91

4–1 MOV Revisited 92
4–2 PUSH/POP 101

4–3 Load-Effective Address 107
4–4 String Data Transfers 109
4–5 Miscellaneous Data Transfer Instructions 115
4–6 Segment Override Prefix 120
4–7 Assembler Details 120
4–8 Summary 131
4–9 Questions and Problems 133

5 ARITHMETIC AND LOGIC INSTRUCTIONS 136

5–1 Addition, Subtraction, and Comparison 137
5–2 Multiplication and Division 148
5–3 BCD and ASCII Arithmetic 154
5–4 Basic Logic Instructions 157
5–5 Shifts and Rotates 164
5–6 String Comparisons 168
5–7 Summary 170
5–8 Questions and Problems 171

6 PROGRAM CONTROL INSTRUCTIONS 175

6–1 The Jump Group 175
6–2 Procedures 185
6–3 Introduction to Interrupts 191
6–4 Machine Control and Miscellaneous Instructions 195
6–5 Summary 199
6–6 Questions and Problems 201

7 PROGRAMMING WITH DOS AND BIOS FUNCTIONS 203

7–1 Modular Programming 204
7–2 Using the Keyboard and Video Display 212
7–3 Data Conversions 223
7–4 VGA Graphic Displays 234
7–5 Disk Files 247
7–6 Summary 267
7–7 Questions and Problems 269

8 THE INTEL FAMILY OF ARITHMETIC COPROCESSORS 273

8–1 Data Formats for the Arithmetic Coprocessor 274
8–2 The 80X87 Architecture 278
8–3 Instruction Set 283
8–4 Programming with the Arithmetic Coprocessor 313

8–5 Summary 321
8–6 Questions and Problems 323

9 THE 80386 AND 80486 MICROPROCESSORS 325

9–1 Introduction to the 80386 Microprocessor 326
9–2 Special 80386 Registers 337
9–3 80386 Memory Management 340
9–4 Moving to Protected Mode 350
9–5 Virtual 8086 Mode 356
9–6 The Memory Paging Mechanism 357
9–7 Introduction to the 80486 Microprocessor 363
9–8 Summary 375
9–9 Questions and Problems 377

10 MEMORY INTERFACE 379

10–1 Memory Devices 380
10–2 Address Decoding 391
10–3 8088 (8-bit) Memory Interface 400
10–4 80386SX (16-bit) Memory Interface 409
10–5 80386DX and 80486 (32-bit) Memory Interface 416
10–6 Pentium (64-bit) Memory Interface 420
10–7 Interleaved Memory 423
10–8 Dynamic RAM 427
10–9 Cache Memory 437
10–10 Summary 439
10–11 Questions and Problems 441

11 BASIC I/O INTERFACE 444

11–1 Introduction to I/O Interfacing 444
11–2 I/O Port Address Decoding 452
11–3 The Programmable Peripheral Interface 459
11–4 The 8279 Programmable Keyboard/Display Interface 476
11–5 8254 Programmable Interval Timer 485
11–6 8251A Programmable Communications Interface 496
11–7 Analog-to-Digital (ADC) and Digital-to-Analog (DAC) Converters 506
11–8 Summary 514
11–9 Questions and Problems 516

12 INTERRUPTS 519

12–1 Basic Interrupt Processing 519
12–2 Hardware Interrupts 530

12–3 Expanding the Interrupt Structure 537
12–4 8259A Programmable Interrupt Controller 541
12–5 Real-Time Clock 554
12–6 Summary 556
12–7 Questions and Problems 557

13 DIRECT MEMORY ACCESS AND DMA-CONTROLLED I/O 559

13–1 Basic DMA Operation 560
13–2 The 8237 DMA Controller 561
13–3 Shared-Bus Operation 577
13–4 Disk Memory Systems 598
13–5 Video Displays 607
13–6 Summary 614
13–7 Questions and Problems 615

14 THE PENTIUM MICROPROCESSOR 617

14–1 Introduction to the Pentium Microprocessor 618
14–2 Special Pentium Registers 628
14–3 Pentium Memory Management 630
14–4 New Pentium Instructions 631
14–5 Summary 633
14–6 Questions and Problems 634

APPENDIXES 636

A The Assembler and Disk Operating System 636
B Instruction Set Summary 685
C Flag Bit Changes 771
D Answers to Selected Even-Numbered Questions and Problems 773

INDEX 799

# CHAPTER 1

# Introduction to the Microprocessor and Computer

## INTRODUCTION

This chapter provides an overview of the microprocessor. Discussions of the history of computers and the function of the microprocessor in microprocessor-based computer systems are included. Terms and jargon of the computer field are introduced so that *computerese* can be understood and used when discussing microprocessors and computers.

The block diagram, and a description of the function of each block, detail the operation of a computer system. The chapter also details the memory and input/output system of the personal computer.

Finally, the way that data are stored in the memory is provided so each data type can be used with software development. Numeric data are stored as integers, floating-point, and binary-coded decimal (BCD), while ASCII (American Standard Code for Information Interchange) code represents alphanumeric data.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Converse using appropriate computer terminology including: bit, byte, data, real memory system, expanded memory system (EMS), extended memory system (XMS), DOS, BIOS, I/O, and so forth.
2. Briefly detail the history of the computer and list applications performed by the computer.
3. Draw the block diagram of a computer system and explain the purpose of each block.
4. Describe the function of the microprocessor and detail its basic operation.
5. Define the contents of the memory system in the personal computer.
6. Differentiate and represent numeric and alphabetic information as integers, floating-point, BCD, and ASCII data.

# 1–1 A HISTORICAL BACKGROUND

This first section outlines the historical events leading to the development of the microprocessor and, specifically, the extremely powerful 80386, 80486, and Pentium microprocessors currently in use. Although a study of history is not essential to understanding the microprocessor, it makes for interesting reading and provides a historical perspective of the evolution of the computer.

## The Mechanical Age

The idea of a computing system is not new—it has been around since long before electrical and electronic devices. The idea of calculating with a machine dates to at least 500 B.C. when the Babylonians invented the *abacus*, the first mechanical calculator. The abacus was used by the ancient Babylonian priesthood to keep track of their vast storehouses of grain. The abacus, which was used extensively and is still in use today, was not improved until 1643, when Blaise Pascal, the mathematician, invented a *calculator* constructed of gears and wheels. Each gear contained ten teeth that, when moved one complete revolution, advanced a second gear one place. This same principal is employed in automobile odometers and is the basis of all mechanical calculators. The *Pascal programming language* is named in honor of Blaise Pascal for his pioneering work in mathematics and with the mechanical calculator.

The first practical, geared, mechanical machines used to automatically compute information date to the early 1800s. This was before humans invented the light bulb or before much was known about electricity. In this dawn of the computer age, humans dreamed of mechanical machines that could compute numerical facts using a program—not merely calculate facts like a calculator.

One early pioneer of mechanical computing machinery was Charles Babbage. Babbage was commissioned in 1823 by the Royal Astronomical Society of Great Britain to produce a programmable calculating machine. This machine was to generate navigational tables for the Royal Navy. He accepted the challenge and began to create what he called his *Analytical Engine*. This engine was a mechanical computer that stored 1,000 twenty-digit decimal numbers and a variable program that could modify the machine to perform various calculating tasks. Input to his engine was through punched cards similar to those used for computers in the 1950s and 1960s. It is assumed that he got the idea of using punched cards from Jacquard, a Frenchman who in 1801 used punched cards as input to the weaving machine that is today called Jacquard's loom. Jacquard's loom wove intricate patterns in the cloth that it produced. The punched cards programmed the loom.

After many years of work, Babbages's dream began to fade when he realized that the machinists of his day were unable to create the mechanical parts needed to complete his work. The analytical engine required more than 50,000 machined parts, which could not be made with enough precision to allow his engine to function reliably.

## The Electrical Age

The late 1800s saw the advent of the electric motor and with it, a multitude of motor-driven adding machines based on Pascal's mechanical calculator. These mechanical calculators

were common pieces of office equipment until well into the early 1970s, when the first small hand-held electronic calculator, introduced by Bomar, appeared. Monroe was also a leading pioneer of electronic calculators, but their desktop, four-function machines were the size of cash registers.

In 1889, Herman Hollerith developed the punched card for storing data. He also developed a mechanical machine—driven by one of the new electric motors—that counted, sorted, and collated information stored on punched cards. The idea of calculating by machine intrigued the United States government so much that Hollerith was commissioned to use his punched-card system to store and tabulate information for the 1890 census.

In 1896, Hollerith formed the Tabulating Machine Company. This company developed a line of machines that used punched cards for tabulation. After a number of mergers, the Tabulating Machine Company became the International Business Machines Corporation, now referred to as IBM. We often refer to the punched cards used in computer systems as *Hollerith cards* in honor of Herman Hollerith. The 12-bit code used on a punched card is called the *Hollerith code*.

Mechanical machines driven by electric motors continued to dominate the information processing world until the advent of the first electronic calculating machine in 1942. Konrad Zuse, a German inventor, developed an electronic calculating computer, the Z3, that was used by the Germans for aircraft and missile design during World War II. Had Zuse been given funding by the German government, he most likely would have developed a much more powerful computer system. Today, Zuse is finally receiving some belated honor for his pioneering work in the area of digital electronics in the 1930s.

It has been recently discovered (through the declassification of British military documents) that the first truly electronic computer was placed into operation in 1943 to break secret German military codes. This first electronic computer system, which used vacuum tubes, was invented by Alan Turing. Turing called his machine *Colossus*, most likely because of its size. A problem with Colossus was that its design allowed it to break secret German military codes, but it could not solve other problems. Colossus was not programmable; it was a fixed-program computer system, which we today call a *special purpose computer*.

The first general purpose, programmable electronic computer system was developed in 1947 at the University of Pennsylvania. This first modern computer was called the *ENIAC* (*Electronics Numerical Integrator and Calculator*). The ENIAC was a huge machine containing more than 17,000 vacuum tubes and 500 miles of wires. This massive machine weighed more than 30 tons, yet performed only about 100,000 operations per second. The ENIAC thrust the world into the age of electronic computers. The ENIAC was programmed by rewiring its circuits—a process that took many days to accomplish. Workers changed electrical connections on plug-boards that looked much like early telephone switchboards. Another problem with the ENIAC was the life of the vacuum tube components, which required almost constant maintenance.

Breakthroughs that followed were the development of the transistor in 1948, followed by the invention of the integrated circuit by Jack Kilby in 1958. The integrated circuit led to the development of digital integrated circuits (RTL or resistor-to-transistor logic) in the 1960s and the first microprocessor in 1971. At this time, Marcian T. Hoff, an engineer at Intel Corporation, developed the 4004 microprocessor—the device that started the microprocessor revolution that continues today.

## The Microprocessor Age

The world's first microprocessor, the Intel 4004, was a 4-bit microprocessor—a programmable controller on a chip that is meager by today's standards. It addressed only 4,096 four-bit wide memory locations. (A *bit* is a binary digit with a value of one or zero. A 4-bit wide memory location is often called a *nibble* or *nybble*.) The 4004 instruction set contained only 45 instructions. It was fabricated with then-current P-channel MOSFET technology that only allowed it to execute instructions at the rate of 50 KIPs (*kilo-instructions per second*). This was slow when compared to the 100,000 instructions executed per second by the 30-ton ENIAC computer in 1946. But the 4004 weighed much less than an *ounce*.

At first, applications abounded for this device. The 4-bit microprocessor debuted in early video game systems and small microprocessor-based control systems. One such early video game, a shuffleboard game, was produced by Balley. The main problems with this early microprocessor were its speed, word width, and memory size. The evolution of the 4-bit microprocessor ended when Intel released the 4040, an updated version of the 4004. The 4040 operated at a higher speed, although it lacked improvements in word width and memory size. Other companies, in particular Texas Instruments (TMS-1000), also produced 4-bit microprocessors. The 4-bit microprocessor still survives in low-end applications such as microwave ovens and small control systems, and is still available from some microprocessor manufacturers. Most calculators are also still based on 4-bit microprocessors that process 4-bit BCD (*binary-coded decimal*) codes.

Later in 1971, realizing that the microprocessor was a commercially viable product, Intel Corporation released the 8008—an extended 8-bit version of the 4004 microprocessor. The 8008 addressed an expanded memory size (16K bytes) and contained additional instructions (a total of 48) that provided an opportunity for its application in more advanced systems. (A *byte* is generally an 8-bit wide binary number and a *K* is 1,024 bytes. Often, memory size is specified in *K bytes*.)

As engineers developed more demanding uses for the 8008 microprocessor, they discovered that its somewhat small memory size, slow speed, and instruction set limited its usefulness. Intel recognized these limitations and, in 1973, introduced the 8080 microprocessor—the first of the modern 8-bit microprocessors. The flood gates opened; the 8080 ushered in the age of the microprocessor. Soon, other companies began to introduce their own versions of the 8-bit microprocessor. Table 1–1 lists several of these early microprocessors and their manufacturers. Of these early microprocessor producers, only Intel and Motorola continue to successfully manufacture newer and improved versions of the

**TABLE 1–1** Early 8-bit microprocessors

| *Manufacturer* | *Part Number* |
|---|---|
| Fairchild | F-8 |
| Intel | 8080 |
| MOS Technology | 6502 |
| Motorola | MC6800 |
| National Semiconductor | IMP-8 |
| Rockwell International | PPS-8 |
| Zilog | Z-8 |

microprocessor. Zilog still manufactures microprocessors, but has decided to concentrate on microcontrollers rather than general-purpose microprocessors.

*What Was Special About the 8080?* Not only could the 8080 address more memory and execute additional instructions, but it executed them ten times faster than the 8008. An addition that took 20 μs on an 8008-based system required only 2.0 μs on an 8080-based system. This enabled the 8080 to execute 500,000 instructions per second, compared to the 50,000 instructions per second executed by the 8008. Also, the 8080 was compatible with TTL (transistor–transistor logic) and the 8008 was not directly compatible. This made interfacing much easier and less expensive. The 8080 also addresses four times more memory (64K bytes) than the 8008 (16K bytes). These improvements of the 8080 are responsible for the continuing saga of the microprocessor. Incidentally, the first personal computer, the MITS Altair 8800 was released in 1974. (Note that the name *8800* was probably chosen to avoid copyright violations with Intel.) The BASIC language interpreter, written for the Altair computer, was developed by a young man named Bill Gates, who would later found Microsoft Corporation.

*The 8085 Microprocessor.* In 1977, Intel Corporation introduced an updated version of the 8080—the 8085. This was to be the last 8-bit general-purpose microprocessor developed by Intel. Although only slightly more advanced than an 8080, the 8085 executes software at an even higher speed. An addition that took 2.0 μs on the 8080 requires only 1.3 μs on the 8085. This enables the 8085 to execute instructions at a rate of 769,230 per second. The main advantages of the 8085 are its internal clock generator, internal system controller, and higher clock frequency. This higher level of component integration reduced the cost and increased the usefulness of the 8085 microprocessor. Intel has sold well over 100 million copies of the 8085 microprocessor, its most successful 8-bit microprocessor. Because the 8085 is also manufactured (*second-sourced*) by many other companies, there are well over 200 million of these microprocessors in existence. Applications which contain the 8085 are still being used and designed and will likely continue to be popular well into the future. Another company that sold 500 million 8-bit microprocessors is Zilog corporation, which produced the Z-80 microprocessor. The Z-80 uses a language code compatible with the 8085, which means that there are well over 700 million microprocessors that execute 8085/Z-80 compatible code!

## The Modern Microprocessor

In 1978, Intel released the 8086 microprocessor and a year or so later, the 8088. Both devices are 16-bit microprocessors, which execute instructions in as little as 400 ns (2.5 MIPs or 2.5 millions of instructions per second). This represented a major improvement over the execution speed of the 8085. In addition, the 8086 and 8088 address 1M byte of memory, 16 times more memory than the 8085. (A *1M-byte memory* contains 1,024K byte-sized memory locations or 1,048,576 bytes.) This higher execution speed and larger memory size allowed the 8086 and 8088 to replace smaller minicomputers in many applications. One other feature found in the 8086/8088 is an instruction cache or queue that prefetches a few instructions before they are executed. The queue speeds the operation of many sequences of instructions and proved to be the basis for the much larger instruction caches found in modern microprocessors.

The increase in memory size and additional instruction capacity of the 8086 and 8088 have led to many sophisticated applications for microprocessors. Improvements to the instruction set included a multiply and divide instruction, which are missing on earlier microprocessors. Also the number of instructions increased from 45 on the 4004, to 246 on the 8085, to well over 20,000 on the 8086 and 8088 microprocessors. Note that these microprocessors are called CISC (*complex instruction set computers*) because of the number and complexity of instructions. The additional instructions ease the task of developing efficient and sophisticated applications even though their number is at first overwhelming and time-consuming to learn. The 16-bit microprocessor also provides more internal register storage space than the 8-bit microprocessor. The additional registers allow software to be written more efficiently.

The 16-bit microprocessor evolved mainly because of the need for larger memory systems. Applications such as spreadsheets, word processors, spelling checkers, and computer-based thesauruses are memory intensive and require more than the 64K bytes of memory found in 8-bit microprocessors. The 16-bit 8086 and 8088 provide 1M byte of memory for these applications. Soon, even 1M byte of memory proved limiting for large spreadsheets and other applications. This led to the introduction of the 80286 microprocessor, an updated 8086, by Intel in 1983.

*The 80286 Microprocessor.* The 80286 microprocessor (also a 16-bit architecture microprocessor) is almost identical to the 8086 and 8088 except it addresses a 16M byte memory system instead of 1M byte. The instruction set of the 80286 is also almost identical to the 8086 and 8088 except for a few additional instructions that manage the extra 15M bytes of memory. The clock speed of the 80286 is increased so it executes some instructions in as little as 250 ns (4.0 MIPs) with the original release 8.0 MHz version. Some changes also occurred in the internal execution of the instructions that led to an eight-fold increase in speed for many instructions when compared to 8086/8088 instructions.

*The 32-bit Microprocessor.* Applications began to demand faster microprocessor speeds, more memory, and wider data paths. This led to Intel's introduction, in 1986, of the 80386. The 80386 represented a major overhaul of the 16-bit 8086–80286 microprocessor's architecture. The 80386 is Intel's first practical 32-bit microprocessor that contains a 32-bit data bus and a 32-bit memory address. (Note that Intel produced an earlier, unsuccessful, 32-bit microprocessor called the iapx-432.) Through these 32-bit buses, the 80386 addresses up to 4G bytes of memory. (*1G* of memory contains 1,024M or 1,073,741,824 locations.) 4G bytes of memory can store an astounding 1,000,000 typewritten double-spaced pages of data. The 80386 is also available in a few modified versions such as the 80386SX, which addresses 16M bytes of memory through a 16-bit data and 24-bit address bus, and the 80386SL/80386SLC, which addresses 32M byte of memory through a 16-bit data and 25-bit address bus. An 80386SLC version contains an internal cache memory that allows it to process data at even higher rates. Recently Intel has released the 80386EX microprocessor. The 80386EX is called an embedded PC containing all the components of the AT class personal computer. The 80386EX also contains 24 lines for input/ouput data, a 26-bit address bus, a 16-bit data bus, a DRAM refresh controller, and programmable chip selection logic.

Applications that require higher microprocessor speeds and large memory systems include software systems that use a GUI or a *graphical user interface*. Modern graphical

displays often contain 256,000 or more picture elements (*pixels or pels*). The least sophisticated VGA (*variable graphics array*) video display has a resolution of 640 pixels per scanning line with 480 scanning lines. In order to display one screen of information, each picture element must be changed. This requires a high-speed microprocessor. Many new software packages use this type of video interface. These GUI-based packages require high microprocessor speeds and, often, accelerated video adapters for quick and efficient manipulation of video text and graphical data. The most striking system, which requires high-speed computing for its graphical display interface, is Microsoft Corporation's Windows.* We often call a GUI a *WYSIWYG* (what you see is what you get) display.

The 32-bit microprocessor is needed because of the size of its data bus, which transfers real (single-precision, floating-point) numbers that require 32-bit wide memory. In order to efficiently process 32-bit real numbers, the microprocessor must efficiently pass them between itself and memory. It takes four read-or-write cycles for them to pass through an 8-bit data bus, but only one read-or-write cycle is required for them to pass through a 32-bit data bus. This significantly increases the speed of any program that manipulates real numbers. Most high-level languages, spreadsheets, and database management systems use real numbers for data storage. Real numbers are also used in graphical design packages that use vectors to plot images on the video screen. These include such CAD (*computer aided drafting/design*) systems as AUTOCAD, ORCAD, and so forth.

Besides providing higher clocking speeds, the 80386 includes a memory management unit that allows memory resources to be allocated and managed by the operating system. Earlier microprocessors left memory management completely to the software. The 80386 includes hardware circuitry for memory management and memory assignment, which improves its efficiency and reduces software overhead.

The instruction set of the 80386 microprocessor is upward compatible with the earlier 8086, 8088, and 80286 microprocessors. Additional instructions reference the 32-bit registers and manage the memory system. Note that memory management instructions and techniques used by the 80286 are also compatible with the 80386 microprocessor. These features allow older, 16-bit software to operate on the 80386 microprocessor.

*The 80486 Microprocessor.* In 1989, Intel released the 80486 microprocessor, which incorporates an 80386-like microprocessor, an 80387-like numeric coprocessor, and an 8K byte cache memory system into one integrated package. Although the 80486 is not radically different from the 80386, it does include one substantial change: the internal structure of the 80486 is modified so about half of its instructions execute in one clock instead of two clocks. Because the 80486 is available in a 50 MHz version, about half the instructions execute in 25 ns (50 MIPs). The average speed improvement for a typical mix of instructions is about 50 percent over the 80386 operated at the same clock speed. Newer versions of the 80486 execute instructions at even higher speeds with a 66 MHz double-clocked version. The double-clocked 66 MHz version executes instructions at the rate of 66 MHz with memory transfers executed at the rate of 33 MHz. A triple-clocked version, from Intel and IBM, improves the internal execution speed to 99 MHz. The future promises to bring microprocessors that internally execute instructions at rates of up to 150 MHz or higher.

---

*Windows is a registered trademark of Microsoft Corporation.

Other versions of the 80486 are called Overdrive* processors. The Overdrive processor is actually a double-clocked version of the 80486DX that replaces the 80486SX or the slower 80486DX. When the Overdrive processor is plugged into its socket, it disables or replaces the 80486SX or 80486DX and functions as a double-clocked version of the microprocessor. For example, if an 80486SX operating at 25 MHz is replaced with an Overdrive microprocessor, it functions as a 80486DX2 50 MHz microprocessor.

Table 1–2 lists many microprocessors produced by Intel and Motorola with information about their word and memory sizes. Other companies produce microprocessors, but none have attained the success of Intel or Motorola.

*The Pentium Microprocessor.* The Pentium, introduced in 1993, is similar to the 80386 and 80486 microprocessors. This microprocessor was originally called the P5 or 80586,

**TABLE 1–2.** Data bus width and memory size of modern microprocessors

| *Manufacturer* | *Part* | *Data Bus Width* | *Memory Size* |
|---|---|---|---|
| Intel | 8048 | 8 | 2K internal |
| | 8051 | 8 | 8K internal |
| | 8085A | 8 | 64K |
| | 8086 | 16 | 1M |
| | 8088 | 8 | 1M |
| | 8096 | 16 | 8K internal |
| | 80186 | 16 | 1M |
| | 80188 | 8 | 1M |
| | 80286 | 16 | 16M |
| | 80386EX | 16 | 64M |
| | 80386DX | 32 | 4G |
| | 80386SL | 16 | 32M |
| | 80386SLC | 16 | 32M + 8K cache |
| | 80386SX | 16 | 16M |
| | 80486DX | 32 | 4G + 8K cache |
| | 80486SX | 32 | 4G + 8K cache |
| | Pentium | 64 | 4G + 16K cache |
| Motorola | 6800 | 8 | 64K |
| | 6805 | 8 | 2K |
| | 6809 | 8 | 64K |
| | 68000 | 16 | 16M |
| | 68008Q | 8 | 1M |
| | 68008D | 8 | 4M |
| | 68010 | 16 | 16M |
| | 68020 | 32 | 4G |
| | 68030 | 32 | 4G + 256 cache |
| | 68040 | 32 | 4G + 8K cache |
| | 68060 | 64 | 4G + 16K cache |

*Overdrive is a registered trademark of Intel Corporation.

but Intel decided not to use a number because it appears to be impossible to copyright a number. The two introductory versions of the Pentium operate with a clocking frequency of 60 MHz and 66 MHz and a speed of 110 MIPs, with higher frequency (100 MHz) versions on the horizon. Another difference is that the cache size is increased to 16K byte from the 8K byte cache found in the 80486. The Pentium contains an 8K byte instruction cache and an 8K byte data cache. This allows a program that transfers a large amount of memory data to still benefit from a cache. The memory system contains up to 4G bytes with the data bus width increased from 32 bits, found in the 80386 and 80486, to a full 64 bits. This wider data bus width accommodates double-precision floating-point numbers used for high-speed vector-generated graphical displays. It also transfers data between the memory system and microprocessor at a higher rate. This should allow *virtual reality*—software that operates at more realistic rates—on current and future Pentium-based platforms. The widened data bus and higher execution speed of the Pentium should also allow full-frame video displays that operate at scan rates of 30 Hz or higher—comparable to commercial television.

Probably the most ingenious feature of the Pentium is its dual integer processors. The Pentium can execute two instructions not dependent on each other simultaneously, because it contains two independent internal-integer processors called *superscaler technology*. This allows the Pentium to execute two instructions per clocking period. Another feature that enhances performance is a jump prediction technology that speeds the execution of programs that include loops. Like the 80486, the Pentium employs an internal floating-point coprocessor to handle floating-point data, albeit at about a five times speed improvement. These features portend continued success for the Intel family of microprocessors. They also may allow the Pentium to replace some of the RISC (*reduced instruction set computer*) machines that currently execute one instruction per clock. Note that some newer RISC processors execute more than one instruction per clock through the introduction of superscaler technology.

In order to compare the speeds of various microprocessors, Intel devised the iCOMP rating index. This index is a composite of SPEC92, ZD Bench, and Power Meter. Figure 1–1 shows the relative speeds of the more commonly used microprocessors.

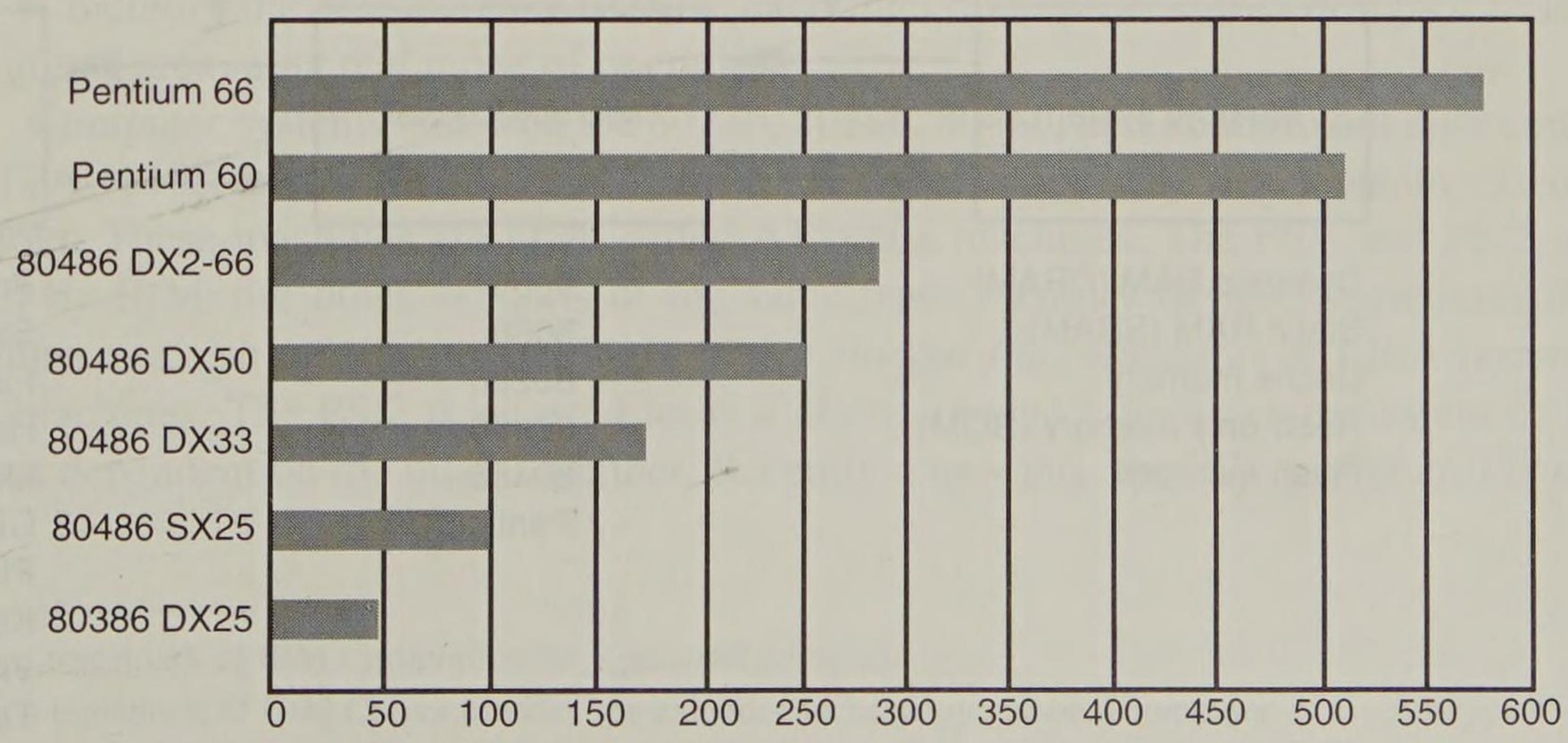



**FIGURE 1–1** The Intel iCOMP Rating Index.

## 1–2 THE MICROPROCESSOR–BASED COMPUTER SYSTEM

The computer system has undergone many changes in recent history. Machines which once filled large rooms have been reduced to small desktop computer systems because of the microprocessor. These desktop computers possess computing power that was only dreamed of a few years ago. Million-dollar mainframe computer systems, developed in the early 1980s, are not as powerful as the 80386, 80486, or Pentium microprocessor-based computers of today. In fact, many smaller companies are replacing their mainframe computers with microprocessor-based systems. The Digital Equipment Corporation and other companies have stopped producing mainframe computer systems in order to concentrate their resources on microprocessor-based computer systems.

This section shows the structure of the microprocessor-based computer system. This structure includes information about the memory and operating system used in many microprocessor-based computer systems.

Refer to Figure 1–2 for the block diagram of a computer system. Notice that the block diagram is composed of three blocks interconnected by buses. (A *bus* is set of common connections that carry the same type of information. For example, the address bus, which contains 20 to 32 connections, is a bus that conveys the memory address to the memory.) These blocks and their function in a personal computer are outlined in this section.

### The Memory and I/O System

The memory structures of all Intel 8086–80486 and Pentium-based computer systems are similar. This includes everything from the first personal computer systems introduced by IBM in 1981, which were based on the 8088, to the most powerful, high-speed versions available today, which are based on the Pentium microprocessor. Figure 1–3 illustrates the

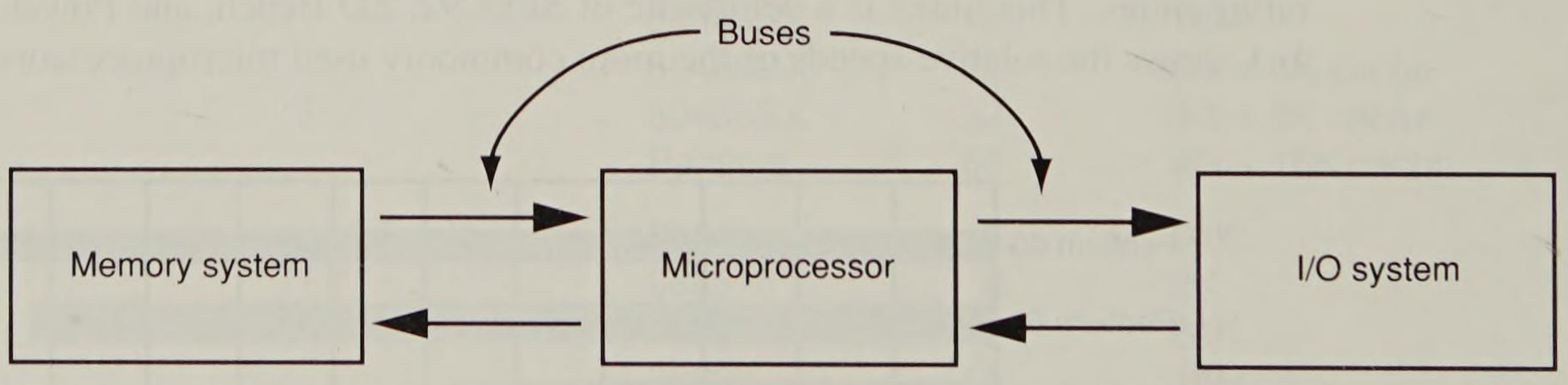



**FIGURE 1–2** The block diagram of a microprocessor-based computer system.

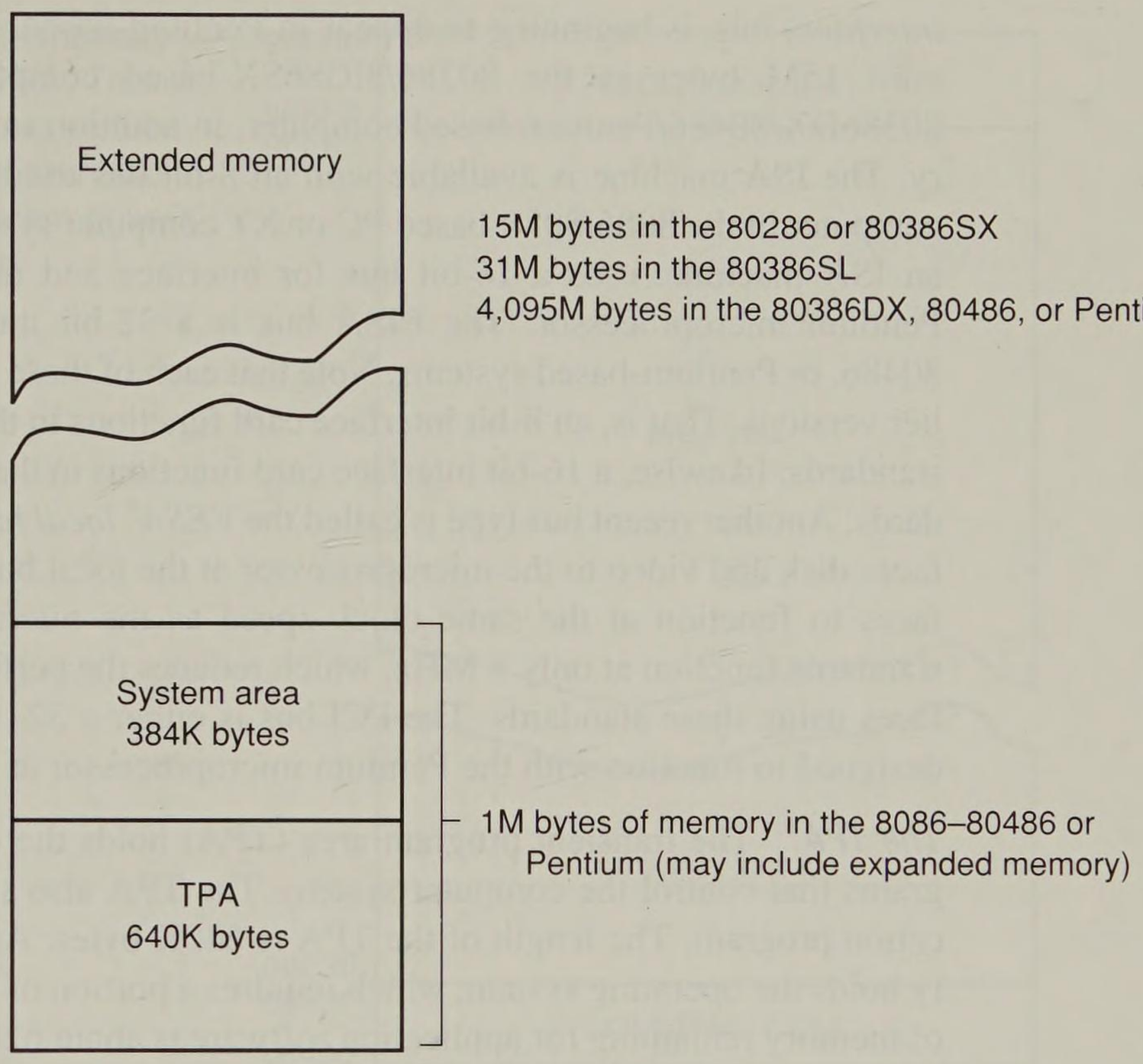



**FIGURE 1–3** The memory map of a personal computer system.

memory map of a personal computer system. This map applies to any IBM personal computer or any of the many IBM compatible clones in existence.

The memory system is divided into three main parts: TPA (*transient program area*), *system area*, and XMS (*extended memory system*). The type of microprocessor in your computer determines whether the extended memory system exists. If the computer is based upon an oider 8086 or 8088 (a PC* or XT†), the TPA and system areas exist, but there is no extended memory area. The PC and XT contain 640K bytes of TPA and 384K bytes of system memory for a total memory size of 1M bytes. We often call the first 1M byte of memory the *real memory* because each Intel microprocessor is designed to function in this area in its real mode of operation.

Computer systems based on the 80286, 80386, 80486, and Pentium not only contain the TPA (640K bytes) and system area (384K bytes), they may also contain extended memory. These machines are often called AT‡ class machines. The PS/1 and PS/2, produced by IBM, are other versions of the same basic memory design. Sometimes these machines are also referred to as ISA (*industry standard architecture*) or EISA (*extended ISA*) machines. The PS/2 is referred to as a *Micro-channel*§ *architecture* system or ISA system dependent on the model number. Recently a new bus, the PCI (*peripheral control*

---

*PC is a trademark of IBM Corporation for a personal computer.

†XT is a trademark of IBM Corporation for the extended technology personal computer.

‡AT is a trademark of IBM Corporation used to denote an advanced class computer system.

§Micro-channel is a registered trademark of IBM Corporation.

*interface*) bus, is beginning to appear in Pentium-based systems. Extended memory contains 15M bytes in the 80286/80386SX-based computer and 4,095M bytes in the 80386DX/80486/Pentium-based computer, in addition to the first 1M byte of real memory. The ISA machine is available with an 8-bit bus used to interface 8-bit devices to the computer in the 8086/8088-based PC or XT computer system. The AT class machine, also an ISA machine, uses a 16-bit bus for interface and may contain an 80286–80486 or Pentium microprocessor. The EISA bus is a 32-bit interface bus found in 80386DX, 80486, or Pentium-based systems. Note that each of these buses is compatible with the earlier versions. That is, an 8-bit interface card functions in the 16-bit ISA or 32-bit EISA bus standards; likewise, a 16-bit interface card functions in the 16-bit ISA or 32-bit EISA standards. Another recent bus type is called the *VESA* local bus* or *VL* bus. The local bus interfaces disk and video to the microprocessor at the local bus level. This allows 32-bit interfaces to function at the same clock speed as the microprocessor. The ISA and EISA standards function at only 8 MHz, which reduces the performance of disk and video interfaces using these standards. The PCI bus is either a 32- or 64-bit bus that is specifically designed to function with the Pentium microprocessor at 60 or 66 MHz.

*The TPA.* The transient program area (TPA) holds the operating system and other programs that control the computer system. The TPA also stores any currently active application program. The length of the TPA is 640K bytes. As mentioned, this area of memory holds the operating system, which requires a portion of the TPA. In practice, the amount of memory remaining for application software is about 628K bytes if MSDOS† version 5.0 or 6.X is used as an operating system. Earlier versions of MSDOS required more of the TPA and often left only 530K bytes or less for application programs. Another operating system found in personal computers is PCDOS.‡ PCDOS and MSDOS are compatible, so either functions in the same manner for application programs. Windows and OS/2§ are other operating systems that are compatible with DOS and allow DOS programs to execute. The *DOS* (disk operating system) controls the way that the disk memory is organized and controlled, as well as the function and control of some of the I/O devices connected to the system. Figure 1–4 shows the organization of the TPA in a computer system.

The memory map shows how the many areas of the TPA are used for system programs, data, and drivers. It also shows a large area of memory available for application programs. To the left of each area is a hexadecimal number that represents the memory addresses that begin and end each data area. Hexadecimal memory addresses or *memory locations* are used to number each byte of the memory system. (A *hexadecimal* number is a number represented in radix 16 or base 16 with each digit representing a value from 0–9 and A–F.)

The *interrupt vectors* access various features of the DOS, BIOS (basic I/O system), and applications. The *BIOS* is a collection of programs stored in a read-only memory (ROM) or flash memory that operate many of the I/O devices connected to your computer system. These programs are stored in the system area defined later in this section of the chapter.

---

*VESA is the Video Electronics Standards Association.

†MSDOS (Microsoft Disk Operating System) is a trademark of Microsoft Corporation.

‡PCDOS (Personal Computer Disk Operating System) is a trademark of IBM Corporation.

§OS/2 (operating system version 2) is a trademark of IBM Corporation.

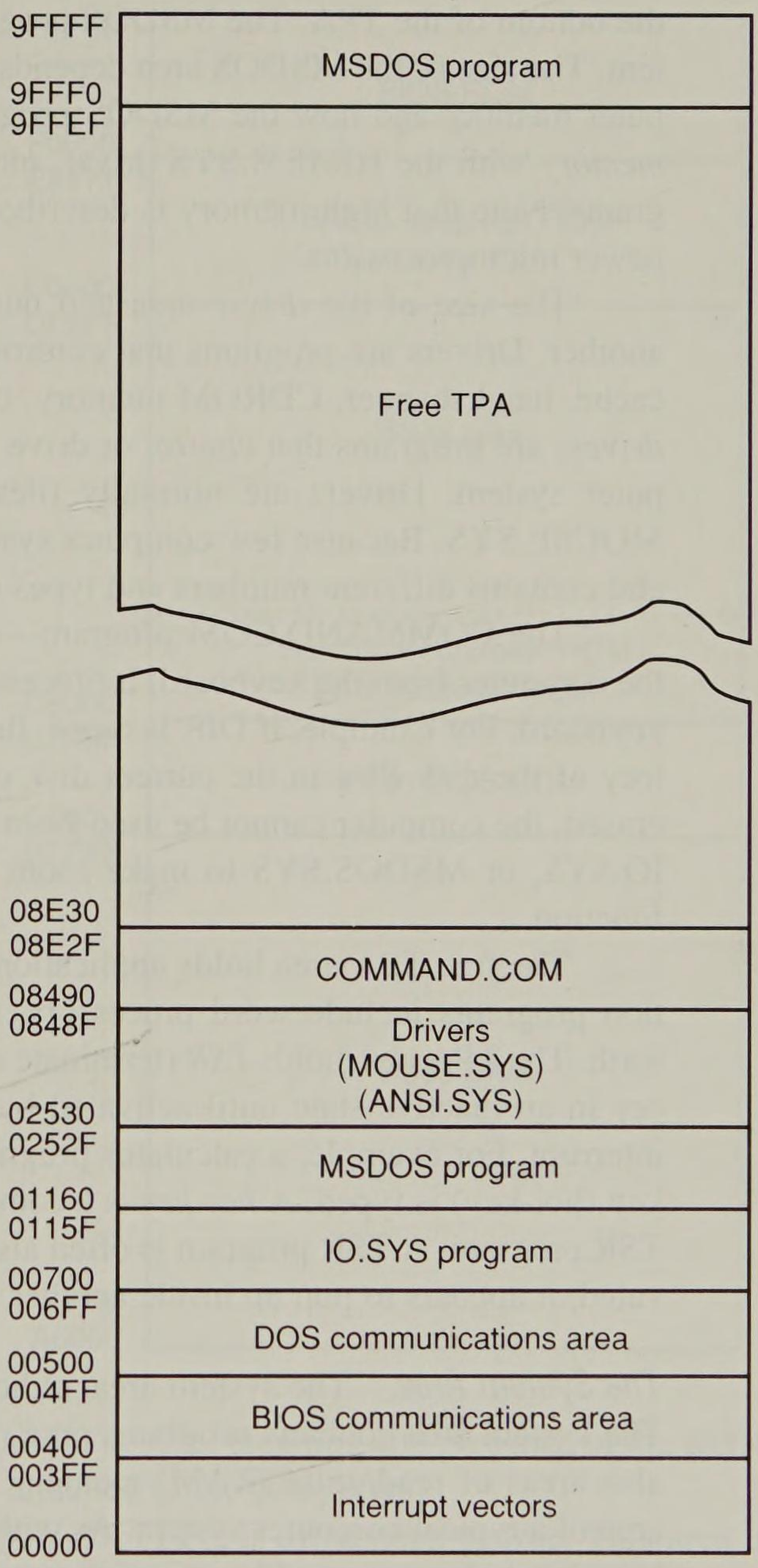



**FIGURE 1–4** The memory map of the DOS TPA. This map varies from one version of DOS to another and also changes with different drivers and configurations.

The BIOS and DOS communications areas contain transient data used by programs to access I/O devices and the internal features of the computer system. These are stored in the TPA so they can be changed as the system operates. Note that the TPA contains read/write memory (called *RAM* or *random access memory*) so it can change as a program executes.

The IO.SYS is a program that loads into the TPA from the disk whenever an MSDOS or PCDOS system is started. The IO.SYS contains programs that allow DOS to use the keyboard, video display, printer, and other I/O devices often found in the computer system. The IO.SYS programs link DOS to the programs stored on the BIOS ROM.

The MSDOS (PCDOS) program occupies two areas of memory. One area is 16 bytes in length and is located at the top of the TPA. The other is much larger and is located near

the bottom of the TPA. The MSDOS program controls the operation of the computer system. The size of the MSDOS area depends on the version of MSDOS installed in the computer memory and how the MSDOS program is installed. If MSDOS is installed in *high memory* with the HIMEM.SYS driver, most of the TPA is freed to hold application programs. Note that high memory is described later in this text and only applies to 80286 or newer microprocessors.

The size of the *driver* area and number of drivers change from one computer to another. Drivers are programs that control installable I/O devices (such as a mouse, disk cache, hand scanner, CDROM memory, or installable drivers) and programs. *Installable drivers* are programs that control or drive devices or programs that are added to the computer system. Drivers are normally files that have an extension of .SYS such as in MOUSE.SYS. Because few computer systems are identical, the driver area varies in size and contains different numbers and types of drivers.

The COMMAND.COM program—command processor—controls the operation of the computer from the keyboard. It processes the DOS commands as they are typed on the keyboard. For example, if DIR is typed, the COMMAND.COM program displays a directory of the disk files in the current disk directory. If the COMMAND.COM program is erased, the computer cannot be used from the keyboard. Never erase COMMAND.COM, IO.SYS, or MSDOS.SYS to make room for other software, or your computer will not function.

The free TPA area holds application programs as they are executed. These application programs include word processors, spreadsheet programs, CAD programs, and so forth. The TPA also holds *TSR* (terminate and stay resident) programs that remain in memory in an inactive state until activated by a hot-key sequence or other event such as an interrupt. For example, a calculator program is a TSR that activates whenever an ALT C key (hot-key) is typed. A *hot-key* is a combination of keys on the keyboard that activate a TSR program. A TSR program is often also called a *pop-up program*, because, when activated, it appears to pop up inside another program.

*The System Area.* The system area, although smaller than the TPA, is just as important. The system area contains programs on a read-only memory (ROM) or flash memory and also areas of read/write (RAM) memory for data storage. Figure 1–5 shows the system area of a typical computer system. As with the map of the TPA, this map also includes the hexadecimal memory addresses of the various areas.

The first area of the system space contains video display RAM and video control programs on ROM or flash memory. This area generally starts at location A0000H and extends to location C7FFFH. The size and amount of memory used depends on the type of video display adapter attached to the system. Display adapters that are often attached to a computer include the earlier CGA (*color graphics adapter*) and EGA (*enhanced graphics adapter*) or one of the many newer forms of VGA (*variable graphics array*). Generally, the video RAM located at A0000H–AFFFFH stores graphical or bit-mapped data and the memory at B0000H–BFFFFH stores text data. The *video BIOS*, located on a ROM or flash memory, is found at locations C0000H–C7FFFH and contains programs that control the video display.

If a hard disk memory is attached to the computer, the interface card might contain a ROM. The ROM, often found with older MFM or RLL hard disk drives, holds low-level

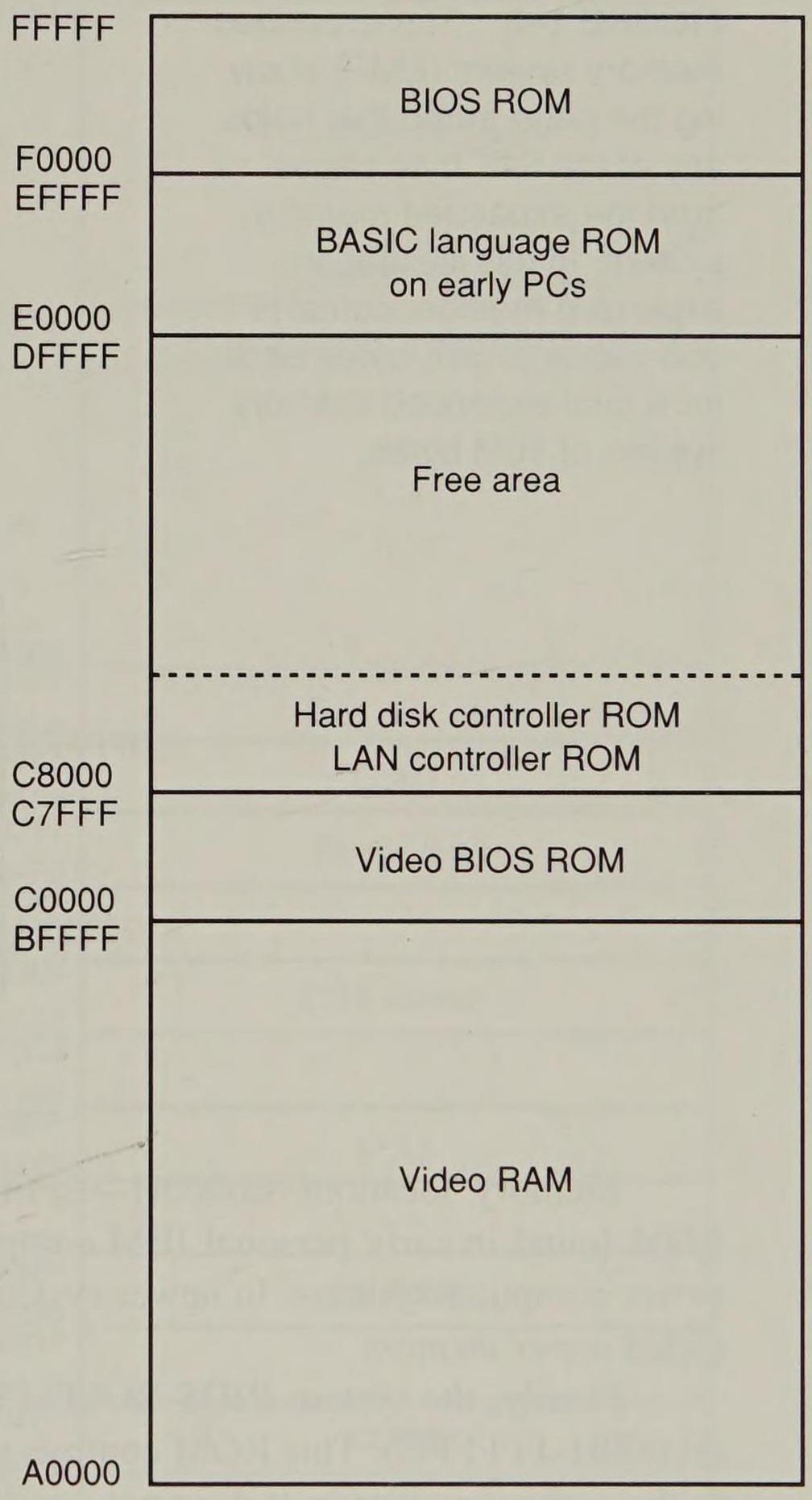



**FIGURE 1–5** The system area of a typical computer system.

format software at location C8000H. The size, location, and presence of the ROM depends on the type of hard disk adapter attached to the computer.

The area at locations C8000H–DFFFFH is often open or free. This area is used for the *expanded memory system* (EMS) in a PC or XT or the *upper memory system* in an AT system. Its use depends on the system and its configuration. The expanded memory system allows a 64K byte page of memory to be used by application programs. This 64K byte page frame (usually locations D0000H–DFFFFH) is expandable by switching other 64K byte pages of memory into this range of memory addresses. Figure 1–6 shows the expanded memory system. Most application programs that state they are LIM 4.0 driver compatible can use expanded memory. The LIM 4.0 memory management driver is the result of Lotus, Intel, and Microsoft standardizing access to expanded memory systems. Note that expanded memory is slow because a change to a new 64K byte memory page requires action by the driver. Also note that expanded memory was designed to expand the memory system of the early 8086/8088-based computer systems. In most cases, except for some DOS-based games that use the sound card, expanded memory should be avoided in 80386/80486/Pentium-based systems.

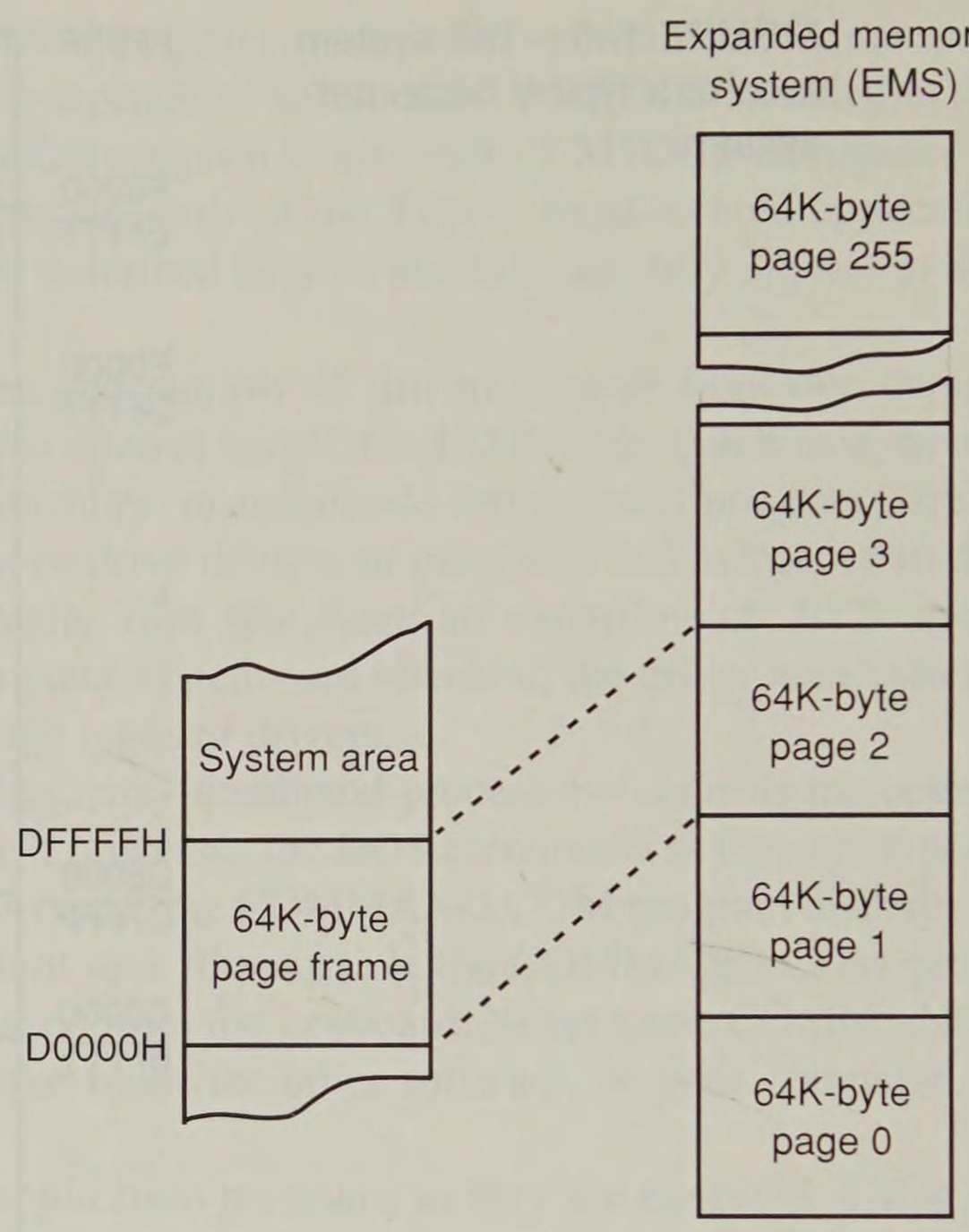



**FIGURE 1–6** The expanded memory system (EMS) showing the page frame that holds any of the 64K byte pages from the expanded memory system. In this illustration, expanded memory contains 256 pages of 64K bytes each, for a total expanded memory system of 16M bytes.

Memory locations E0000H—EFFFFH contain the cassette BASIC language on ROM found in early personal IBM computer systems. This area is often open or free in newer computer systems. In newer systems, we often back-fill this area with extra RAM called *upper memory*.

Finally, the system BIOS ROM is located in the top 64K bytes of the system area (F0000H–FFFFFH). This ROM controls the operation of the basic I/O devices connected to the computer system. It does not control the operation of the video system, which has its own BIOS ROM at location C0000H. The first part of the BIOS (F0000H–F7FFFH) contains programs that set up the computer and the second part contains procedures that control the basic I/O system. Once the system is set up, we can often use F0000H–F7FFFH for upper memory blocks if EMM386.EXE is installed.

*I/O Space.* The I/O (*input/output*) space in a computer system extends from I/O port 0000H to port FFFFH. (An *I/O port address* is similar to a memory address except that instead of addressing memory, it addresses an I/O device.) The *I/O devices* allow the microprocessor to communicate between itself and the outside world. The I/O space allows the computer to access up to 64K different I/O devices. A great number of these locations are available for expansion in most computer systems. Figure 1–7 shows the I/O map found in many 8086–80486 and Pentium-based personal computer systems.

The I/O area contains two major sections. The area below I/O location 0500H is considered reserved for system devices (see Figure 1–7). The remaining I/O space, from I/O port 0500H through FFFFH, is available for expansion.

Various I/O devices that control the operation of the system are usually not directly addressed. Instead, the BIOS ROM addresses these basic devices, which can vary slightly

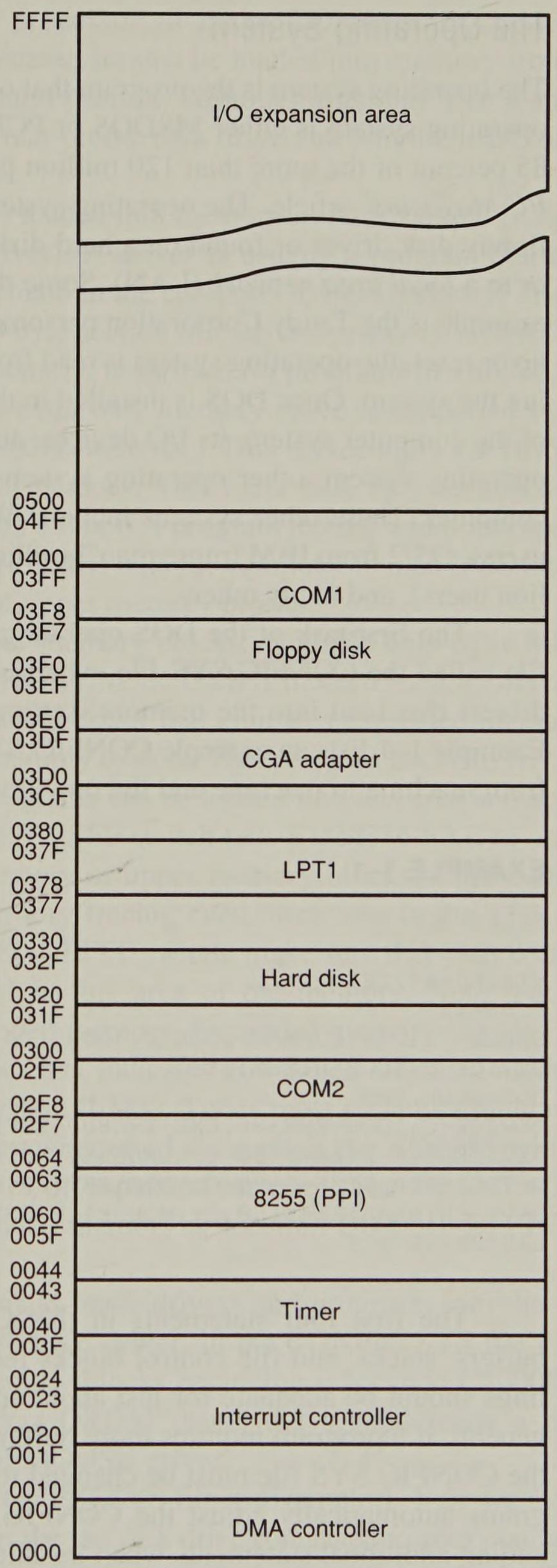



**FIGURE 1–7** The I/O map of a personal computer illustrating many of the I/O areas.

in location and function from one computer to the next. Access to I/O devices should always be made through DOS or BIOS function calls to maintain compatibility from one computer system to another. This map is provided as a guide to illustrate the I/O space in the system.

## The Operating System

The operating system is the program that operates the computer. This text assumes that the operating system is either MSDOS or PCDOS, which are the operating systems found in 85 percent of the more than 120 million personal computers in use, according to a recent *PC Magazine*[*] article. The operating system is stored on a disk that is placed in one of the floppy disk drives or found on a hard disk drive which is either resident to the computer or to a *local area network* (LAN). Some dedicated systems store the DOS on a ROM. An example is the Tandy Corporation personal computer. Each time the computer is powered up or reset, the operating system is read from the disk or LAN. We call this operation *booting* the system. Once DOS is installed in the memory by the boot, it controls the operation of the computer system, its I/O devices, and application programs. In addition to the DOS operating system, other operating systems are sometimes used to control or operate the computer. These other systems include: Windows from Microsoft (more than 11 million users), OS/2 from IBM (more than 2 million users), Unix from AT&T (more than 1.8 million users), and many others.

The first task of the DOS operating system, after loading into memory, is to use a file called the CONFIG.SYS file, which should not be erased. This file specifies various drivers that load into the memory, setting up or *configuring* the machine for operation. Example 1–1 lists an example CONFIG.SYS file. Note that the statements in this file vary from machine to machine and the one illustrated is just an example.

**EXAMPLE 1–1**

```
FILES=30
BUFFERS=30
STACKS=64,128
FCBS=48
SHELL=C:\DOS\COMMAND.COM C:\DOS\ /E:256 /P
DEVICE=C:\DOS\HIMEM.SYS
DOS=HIGH,UMB
DEVICE=C:\DOS\EMM386.EXE I=C800-EFFF NOEMS
DEVICEHIGH SIZE=1EB0 C:\LASERLIB\SONY_CDU.SYS
DEVICEHIGH SIZE=0190 C:\DOS\SETVER.EXE
DEVICEHIGH SIZE=3150 C:\MOUSE1\MOUSE.SYS
LASTDRIVE = F
```

The first four statements in this CONFIG.SYS file set up the number of files, buffers, stacks, and file control blocks required to execute various programs. These settings should be adequate for just about any program loaded into memory using DOS. In general, if a program requires more buffers and so forth, the documentation indicates that the CONFIG.SYS file must be changed to reflect this increased need. Many modern programs automatically adjust the CONFIG.SYS file by changing these parameters or by adding additional statements when installed.

The SHELL command specifies the command processor used with DOS. In this example the COMMAND.COM file is the command processor (also selected by default) using the E:256 /P switches. The E:256 switch sets the environment size to 256 bytes. The

---

[*] *PC Magazine* is a Ziff-Davis publication.

/P switch tells the command processor to make COMMAND.COM permanent. If COMMAND.COM is not permanent, it must be loaded into memory from the disk each time DOS returns to the command prompt. Although this may free a small amount of memory, it increases wear and tear on the disk drive and lengthens the time required to return to the DOS prompt.

The first DEVICE (driver) loaded into memory in this example is a program called HIMEM.SYS (high memory driver). A *driver* is usually a program that controls an I/O device or a program that must remain in the computer system memory. The HIMEM.SYS program allows a 64K minus 16 byte section of extended memory (100000H–10FFEFH), just above the first 1M byte of memory, to be used for programs in a 80286, 80386, 80486, or Pentium-based system. (This extra 64K memory space is supported by most of these computer systems, but not all 80286 systems.) This driver allows a DOS-based system access to 1M plus 64K bytes of memory. This extra 64K byte section of *high memory* holds most of the MSDOS version 5.0 or 6.X program freeing additional space in the TPA. The next command (DOS=HIGH,UMB) tells the computer to load DOS into this high part of memory and allows the use of upper memory blocks.

In order to enable the upper memory blocks, available only in an 80386, 80486, or Pentium-based system, we load the EMM386.EXE *(extended memory manager)* program. The extended memory manager is a driver that emulates expanded memory in extended memory and also the extended memory system. This program backfills free areas of memory within the system area so programs can be loaded into this area and accessed directly by DOS applications. The I=C800–EFFF switch tells EMM386.EXE to use memory area C8000H–EFFFFH for upper memory or upper memory blocks *(UMB)*. Drivers and programs are loaded into upper memory freeing even more area in the TPA for application programs. Before using the I=C800–EFFF switch, make sure that your computer does not contain any system ROM/RAM in this area of the memory. Note that NOEMS tells EMM386.EXE to exclude expanded memory. Expanded memory can also be installed by replacing NOEMS with a number that indicates how much extended memory to allocate to LIM 4.0 expanded memory system (EMS). Today most systems should not use expanded memory. If expanded memory is required we replace the NOEMS with, for example, RAM 1024 to enable 1,024 bytes of expanded memory. The FRAME=D000 statement places the page frame for expanded memory at location D0000H–DFFFFH if expanded memory is enabled.

The DEVICEHIGH command loads drivers and programs into the upper memory blocks allocated by the EMM386.EXE driver. In the CONFIG.SYS file illustrated, three drivers are loaded into upper memory blocks beginning at location C8000H. The first is a program that operates a SONY CDROM drive, the second loads a program called SETVER, and the third loads the MOUSE driver.

The last statement in the CONFIG.SYS file in Example 1–1 shows the LASTDRIVE statement. This tells DOS which is the last disk drive connected to your computer system. By using the LASTDRIVE statement more memory can be freed for use in the TPA. Each drive requires a buffer area and if you use the actual last drive with this statement, extra memory is made available. Other drivers may also be loaded into memory using the CONFIG.SYS file such as a PRINT.SYS driver, ANSI.SYS driver, or any other program that functions as a driver. Driver programs normally contain the DOS extension .SYS to indicate a system file. Be sure to be very careful when changing the CONFIG.SYS file, because an error locks up the

computer system (except for DOS 6.X, which can exit this type of system lockup). Once the computer is locked up by a CONFIG.SYS error the only way to recover is to boot off a DOS floppy disk that contains the operating system with a functioning CONFIG.SYS file.

Once the operating system completes its configuration as dictated by CONFIG.SYS, the AUTOEXEC.BAT *(automatic execution batch)* file is executed by the computer. If none exists, the computer asks for the time and date. Example 1–2 shows a typical AUTOEXEC.BAT file. This is only an example and often variations occur from system to system. The AUTOEXEC.BAT file contains commands that execute when power is first applied to the computer. These are the same commands that could be typed from the keyboard, but the AUTOEXEC.BAT saves us from doing so each time the computer is powered up.

**EXAMPLE 1–2**

```
PATH C:\DOS;C:\;C:\MASM\BIN;C:\MASM\BINB\;C:\UTILITY
PATH C:\WS;C:\LASERLIB
SET BLASTER=A220 I7 D1 T3
SET INCLUDE=C:\MASM\INCLUDE\
SET HELPFILES=C:\MASM\HELP\*.HLP
SET INIT=C:\MASM\INIT\
SET ASMEX=C:\MASM\SAMPLES\
SET TMP=C:\MASM\TMP
SET SOUND=C:\SB
LOADHIGH C:\LASERLIB\MSCDEX.EXE /D:SONY_001 /L:F /M:10
LOADHIGH C:\LASERLIB\LLTSR.EXE ALT-Q
LOADHIGH C:\DOS\FASTOPEN C:=256
LOADHIGH C:\DOS\DOSKEY /BUFSIZE=1024
LOADHIGH C:\LASERLIB\PRINTF.COM
DOSKEY GO=DOSSHELL
DOSSHELL
```

The PATH statement specifies the search paths whenever a program name is typed at the command line. The order of the path search is the same as the order of the paths in the path statement. For example, if PROG is typed at the command line, the machine first searches C:\DOS, then the root directory C:\, then C:\MASM\BIN and so forth until the program named PROG is found. If it isn't found, the command interpreter (COMMAND.COM) informs the user that the program is not found.

The SET statement sets a variable name to a path. This allows names to be associated with paths for batch programs. It's also used to set command strings (environments) for various programs. The first SET command sets the environment for the sound blaster card. The second SET command sets INCLUDE to the path C:\MASM\INCLUDE\. Note that the SET statements are stored in the DOS environment space that was reserved in the CONFIG.SYS file using the SHELL statement. If the environment becomes too large, you must change the SHELL statement to allow more space.

LOADHIGH or LH places programs into upper memory blocks defined by EMM386.EXE program. LOADHIGH is used at any DOS command prompt for loading a program into the high *(upper)* memory area, as long as the computer is an 80386, 80486, or Pentium. The last command in this AUTOEXEC.BAT file is DOSSHELL. The DOSSHELL program is a menu program included with MSDOS version 5.0 or 6.X. This last command

is replaced by WIN, WIN /S (standard mode), WIN /R (real mode in Windows 3.0), or WIN /3 (enhanced mode) to run Microsoft Windows in place of DOSSHELL.

Once the CONFIG.SYS and AUTOEXEC.BAT files are executed, the program name last shown in the AUTOEXEC.BAT file is executed. In this example, the system operates from the DOSSHELL program.

## The Microprocessor

At the heart of the microprocessor-based computer system is the microprocessor integrated circuit. The *microprocessor* is the controlling element in a computer system and is sometimes referred to as the CPU *(central processing unit)*. The microprocessor controls memory and I/O through a series of connections called *buses*. Buses select an I/O or memory device, transfer data between an I/O device or memory and the microprocessor, and control the I/O and memory systems. Memory and I/O are controlled through instructions that are stored in the memory and executed by the microprocessor.

The microprocessor performs three main tasks for the computer system: (1) data transfer between itself and the memory or I/O systems, (2) simple arithmetic and logic operations, and (3) program flow via simple decisions. Although these are simple tasks, it is through them that the microprocessor performs virtually any series of operations or tasks.

The power of the microprocessor is in its ability to execute millions of instructions per second from a *program* or *software* (group of instructions) stored in the memory system. This *stored program concept* has made the microprocessor and computer system a very powerful device. Recall that Babbage wanted to use the stored program concept in his analytical engine.

Table 1–3 shows the arithmetic and logic operations that the 8086–80486 and Pentium microprocessors execute. These operations are very basic, but through them, very complex problems are solved. Data is operated upon from the memory system or internal registers. Data widths are variable and include a *byte* (8 bits), *word* (16 bits), and *doubleword* (32 bits). Note that only the 80386, 80486, and Pentium directly manipulate 8-, 16-, and 32-bit numbers. The earlier 8086–80286 directly manipulate 8- and 16-bit numbers, but not 32-bit numbers. The 80486DX and Pentium also contain a numeric coprocessor that allows them to perform complex arithmetic using floating-point arithmetic.

**TABLE 1–3** Simple arithmetic and logic operations

| *Operation* | *Comment* |
|---|---|
| Addition | |
| Subtraction | |
| Multiplication | |
| Division | |
| AND | Logical multiplication |
| OR | Logical addition |
| NOT | Logical inversion |
| NEG | Arithmetic negation |
| Shift | |
| Rotate | |

Another feature that makes the microprocessor powerful is its ability to make simple decisions. Decisions are based upon numerical facts. For example, a microprocessor can decide whether a number is zero, whether it is positive, and so forth. Through these simple decisions, the microprocessor can modify the program flow so that programs appear to think. Table 1–4 lists the decision-making abilities of the 8086–80486 and Pentium microprocessors.

*Buses.* A *bus* is a common group of wires that interconnect components in a computer system. The buses that interconnect the sections of a computer system transfer address, data, and control information between the microprocessor and its memory and I/O systems. In the microprocessor-based computer system, three buses exist for this transfer of information: address, data, and control. Figure 1–8 shows how these buses interconnect various system components such as the microprocessor, read/write memory (RAM), read-only memory (ROM), and a few I/O devices.

The address bus requests a memory location from the memory or an I/O location from the I/O devices. If I/O is addressed, the address bus contains a 16-bit I/O address of 0000H–FFFFH. The 16-bit I/O address or port number selects one of 64K different I/O devices. If memory is addressed, the address bus contains a memory address. The memory address varies in width with the different versions of the 8086–80486 and Pentium microprocessors. The 8086 and 8088 address 1M byte of memory using a 20-bit address that selects locations 00000H–FFFFFH. The 80286 and 80386SX address 16M bytes of memory using a 24-bit address that selects locations 000000H–FFFFFFH. The 80386SL/80386SLC address 32M bytes of memory using a 25-bit address that selects locations 0000000H–1FFFFFFH. The 80386DX, 80486SX, and 80486DX address 4G bytes of memory using a 32-bit address that selects locations 00000000H–FFFFFFFFH. The Pentium also addresses 4G bytes of memory, but it uses a 64-bit data bus.

The data bus transfers information between the microprocessor and its memory and I/O address space. Data transfers vary in size from 8 bits wide to 64 bits wide in various members of the 8086–80486/Pentium family. The 8088 contains an 8-bit data bus that transfers 8 bits of data at a time. The 8086, 80286, 80386SL, and 80386SX transfer 16-bit data through their data buses; the 80386DX, 80486SX, and 80486DX transfer 32-bit data; and the Pentium transfers 64-bit data. The advantage of a wider data bus is speed in applications that use wide data. For example, if a 32-bit number is stored in memory, it takes the 8088 microprocessor four transfer operations to complete because its data bus is only 8 bits wide.

**TABLE 1–4** Decisions found in an 8086–80486 and Pentium microprocessor

| *Decision* | *Comment* |
|---|---|
| Zero | Test a number for zero or not-zero |
| Sign | Test a number for positive or negative |
| Carry | Test for a carry after an addition or a borrow after a subtraction |
| Parity | Test a number for the count of the number of ones expressed as even or odd |
| Overflow | Test for an invalid signed result |

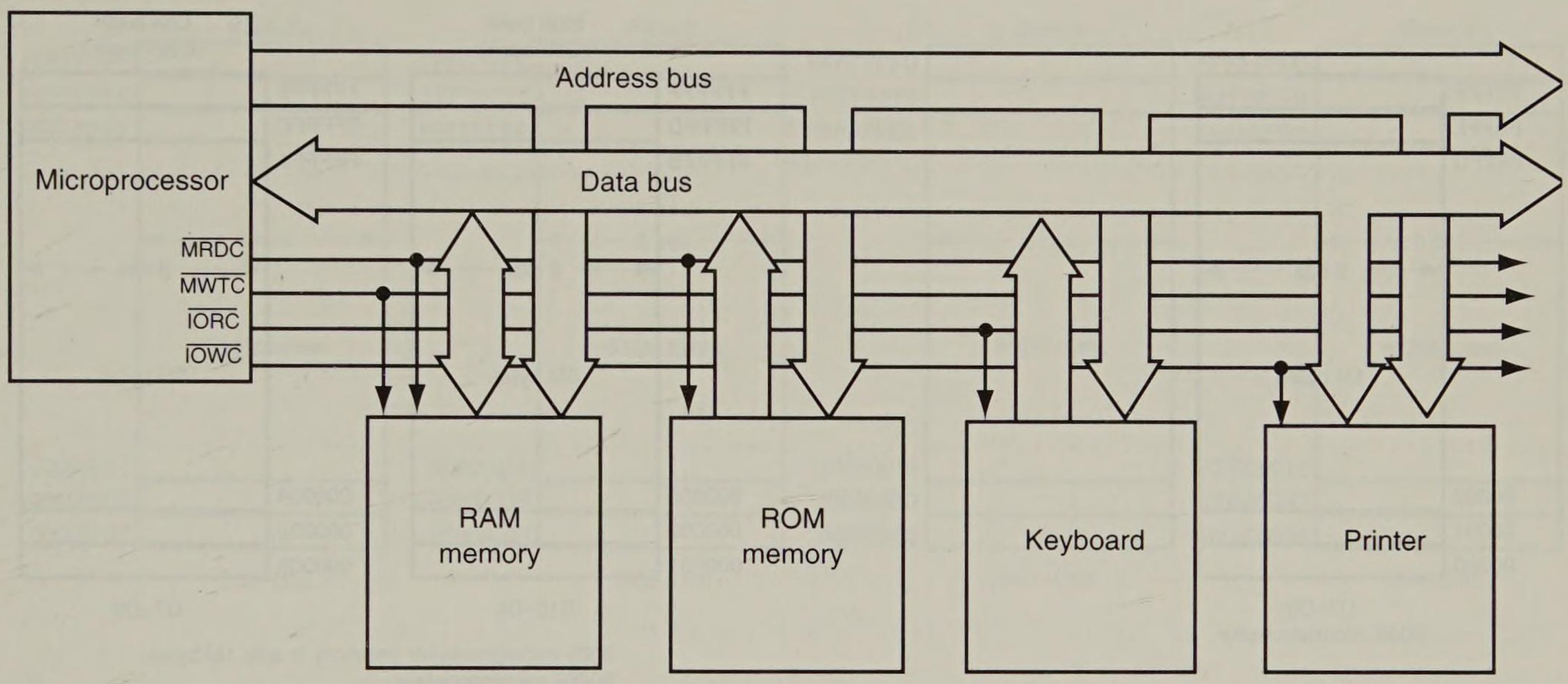



**FIGURE 1–8** The block diagram of a computer system showing the address, data, and control buses. Note that decoders are not shown in this illustration. Note that $\overline{\text{MRDC}}$ (memory read), $\overline{\text{MWTC}}$ (memory write), $\overline{\text{IORC}}$ (I/O read), and $\overline{\text{IOWC}}$ (I/O write) comprise the control bus.

The 80486 accomplishes the same task with one transfer because its data bus is 32 bits wide. Figure 1–9 shows the memory widths and sizes of the 8086–80486 and Pentium microprocessors. Notice how the memory sizes and organizations differ between various members of the 8086–80486 and Pentium microprocessor family. In all family members, the memory is numbered by byte. Notice that the Pentium contains a 64-bit wide data bus.

The control bus contains lines that select the memory or I/O and cause them to perform a read or a write operation. In most computer systems, there are four control bus connections: $\overline{\text{MRDC}}$ *(memory read control)*, $\overline{\text{MWTC}}$ *(memory write control)*, $\overline{\text{IORC}}$ *(I/O read control)*, and $\overline{\text{IOWC}}$ *(I/O write control)*. Note that the overbar indicates that the control signal is active-low; that is, it is active when a logic zero appears on the control line. For example, if $\overline{\text{IOWC}} = 0$, the microprocessor is writing data from the data bus to an I/O device whose address appears on the address bus.

The microprocessor reads the contents of a memory location by sending the memory an address through the address bus. Next it sends the memory read control signal ($\overline{\text{MRDC}}$) to cause memory to read data. Finally the data read from the memory is passed to the microprocessor through the data bus. Whenever a memory write, I/O write, or I/O read occurs, the same sequence ensues, except that different control signals are issued and the data flow out of the microprocessor through its data bus for a write operation.

## 1–3 COMPUTER DATA FORMATS

Successful programming requires a precise understanding of data formats. In this section, we describe the common—but not all possible—data formats used with the 8086–80486

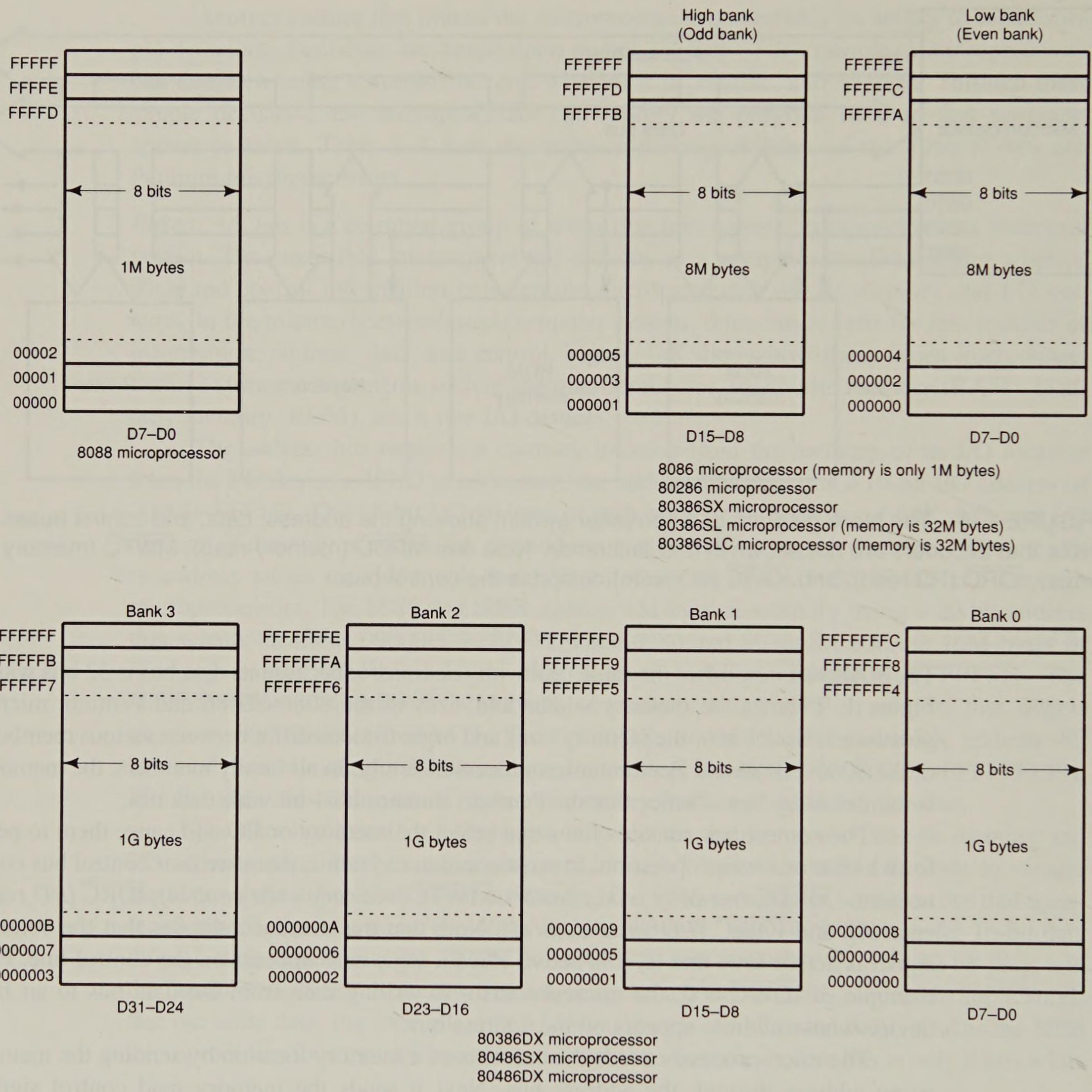



**FIGURE 1–9** The physical memory systems of the 8086 and Pentium microprocessors.

and Pentium microprocessors. Commonly, data appear as ASCII, BCD, signed and unsigned integers, and floating-point numbers (real numbers). Other forms are available, but are not presented here because they are not commonly found.

## ASCII Data

ASCII (*American Standard Code for Information Interchange*) data (see Table 1–5) represents alphanumeric characters in the memory of a computer system. The standard *ASCII*

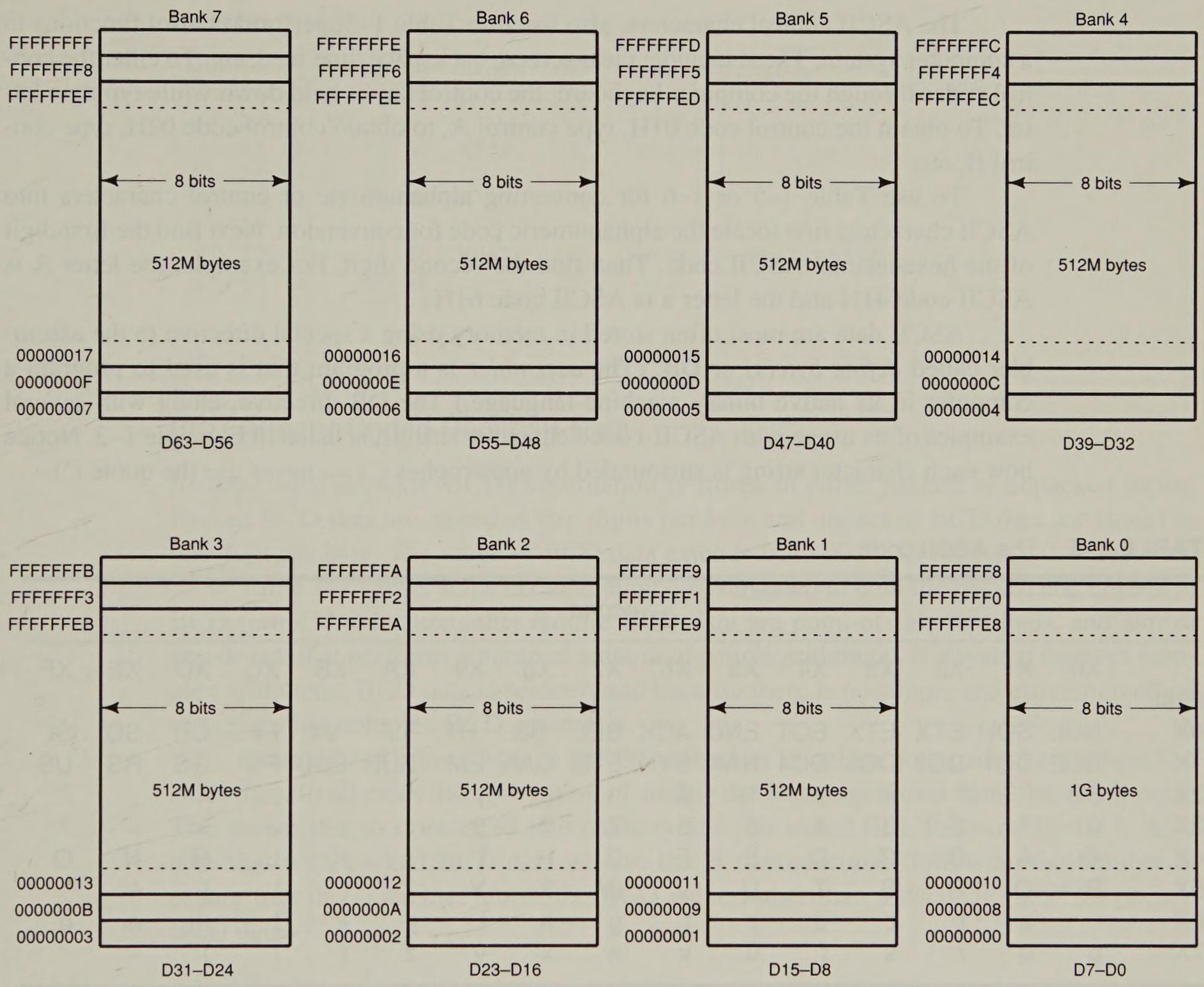



**FIGURE 1–9** (*continued*)

*code* is a 7-bit code with the eighth and most significant bit used to hold parity in some systems. If ASCII data are used with a printer, the most significant bit is a 0 for alphanumeric printing, and 1 for graphics printing. In the personal computer, an extended ASCII character set is selected by placing a logic 1 in the left-most bit. Table 1–6 shows the extended ASCII character set using code 80H–FFH. The extended ASCII characters store some foreign letters and punctuation, Greek characters, mathematical characters, box drawing characters, as well as other special characters. Note that extended characters can vary from one printer to another. The list provided is designed to be used with the IBM ProPrinter* and also matches the special character set found with some word processors.

*ProPrinter is a trademark of the IBM Corporation.

The ASCII control characters, also listed in Table 1–5, perform control functions in a computer system. These include: clear screen, backspace, line feed, etc. To enter the control codes through the computer keyboard, the control key is held down while typing a letter. To obtain the control code 01H, type control A, to obtain control code 02H, type control B, etc.

To use Table 1–5 or 1–6 for converting alphanumeric or control characters into ASCII characters first locate the alphanumeric code for conversion. Next find the first digit of the hexadecimal ASCII code. Then find the second digit. For example, the letter A is ASCII code 41H and the letter a is ASCII code 61H.

ASCII data are most often stored in memory using a special directive to the assembler called *define byte(s)* or DB. (The *assembler* is a program that is used to program a computer in its native binary machine language.) The DB directive, along with several examples of its usage with ASCII-coded character strings, is listed in Example 1–3. Notice how each character string is surrounded by apostrophes (') — never use the quote (‘).

**TABLE 1–5** The ASCII code

| | *Second* | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | XA | XB | XC | XD | XE | XF |
| First | | | | | | | | | | | | | | | | |
| 0X | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1X | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2X | SP | ! | “ | # | $ | % | & | ’ | ( | ) | * | + | , | - | . | / |
| 3X | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4X | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5X | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6X | ‘ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7X | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | |

**TABLE 1–6** Extended ASCII characters (as printed by the IBM Proprinter)

| | *Second* | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8 | X9 | XA | XB | XC | XD | XE | XF |
| First | | | | | | | | | | | | | | | | |
| 0X | | | | ♥ | ◆ | ♣ | ♠ | | | | | | | | | |
| 1X | | | | | ¶ | § | | | | | → | | | | | |
| 8X | Ç | ü | é | â | ä | à | å | ç | ê | ë | è | ï | î | ì | Ä | Å |
| 9X | É | æ | Æ | ô | ö | ò | û | ù | ÿ | Ö | Ü | ¢ | £ | ¥ | ₧ | ƒ |
| AX | á | í | ó | ú | ñ | Ñ | ª | º | ¿ | ⌐ | ¬ | ½ | ¼ | ¡ | « | » |
| BX | ▒ | ▓ | ▓ | \| | ┤ | ╡ | ╢ | ╖ | ╕ | ╜ | ╛ | ┐ | └ | ┴ | ┬ | ├ |
| CX | ┘ | ┌ | ┴ | ┬ | ├ | ─ | ┼ | ╞ | ╟ | ╚ | ╔ | ╩ | ╦ | ╠ | ═ | ╬ |
| DX | ╧ | ╨ | ╤ | ╥ | ╙ | ╘ | ╒ | ╓ | ╫ | ╪ | ┘ | ┌ | ▒ | ▓ | ▓ | ▓ |
| EX | α | ß | Γ | π | Σ | σ | µ | γ | Φ | θ | Ω | δ | ∞ | ø | ∈ | ∩ |
| FX | ≡ | ± | ≥ | ≤ | ⌠ | ⌡ | ÷ | ≈ | ° | • | · | √ | ⁿ | ² | ▪ | |

**EXAMPLE 1–3**

```
0000 42 61 72 72 79  NAMES  DB   'Barry B. Brey'
     20 42 2E 20 42
     72 65 79
000D 57 68 65 72 65  MESS   DB   'Where can it be?'
     20 63 61 6E 20
     69 74 20 62 65
     3F
001D 57 68 61 74 20  WHAT   DB   'What is on first.'
     69 73 20 6F 6E
     20 66 69 72 73
     74 2E
```

## BCD (Binary-Coded Decimal) Data

*Binary-coded decimal* (BCD) information is stored in either packed or unpacked forms. Packed BCD data are stored as two digits per byte and unpacked BCD data are stored as one digit per byte. The range of BCD data extends from $0000_2$ to $1001_2$, or 0–9 decimal.

Table 1–7 shows some decimal numbers converted to both the packed and unpacked BCD forms. Applications that require BCD data are point-of-sale terminals, and almost any device that performs a minimal amount of simple arithmetic. If a system requires complex arithmetic, BCD data is seldom used because there is no simple and efficient method of performing complex BCD arithmetic.

Example 1–4 shows how to use the assembler to define both packed and unpacked BCD data. In all cases the convention of storing the least-significant data first is followed. This means that to store an 83 into memory, the 3 is stored first, followed by the 8. Also note that with packed BCD data we use the H (hexadecimal) following the number to ensure that the assembler stores the BCD value rather than a decimal value for packed BCD data.

**EXAMPLE 1–4**

```
                        ;Unpacked BCD data
                        ;
0000 03 04 05           NUMB1  DB   3,4,5          ;defines the number 534
0003 07 08              NUMB2  DB   7,8            ;defines the number 87
                        ;
                        ;Packed BCD data
                        ;
0005 34 37              NUMB3  DB   34H,37H        ;defines the number 3734
0007 78                 NUMB4  DB   78H            ;defines the number 78
```

**TABLE 1–7** Packed and unpacked BCD data

| *Decimal* | *Packed* | | *Unpacked* | | |
|---|---|---|---|---|---|
| 12 | 0001 0010 | | 0000 0001 | 0000 0010 | |
| 623 | 0000 0110 | 0010 0011 | 0000 0110 | 0000 0010 | 0000 0011 |
| 910 | 0000 1001 | 0001 0000 | 0000 1001 | 0000 0001 | 0000 0000 |

## Byte-Sized Data

Byte-sized data are stored as unsigned and signed integers. Figure 1–10 illustrates both the unsigned and signed forms of the byte-sized integer. The difference in these forms is the weight of the left-most bit position. Its value is 128 for the unsigned integer and –128 for the signed integer. In the signed integer format, the left-most bit represents the sign bit of the number as well as a weight of –128. For example, an 80H represents a value of 128 as an unsigned number and as signed number, it represents a value of –128. Unsigned integers range in value from 00H–FFH (0–255). Signed integers range in value from –128 to 0 to +127.

Although negative signed numbers are represented in this way, they are stored in the *two's complement form*. The method of evaluating a signed number, using the weights of each bit position, is much easier than the act of two's complementing a number to find its value. This is especially true in the world of calculators designed for programmers.

Whenever a number is two's complemented, its sign changes from negative to positive or positive to negative. For example, the number 00001000 is a +8. Its negative value (–8) is found by two's complementing the +8. To form a two's complement, we first one's complement the number. To one's complement a number, invert each bit of a number from zero to one or from one to zero. Once the one's complement is formed, the two's complement is found by adding a one to the one's complement. Example 1–5 shows how numbers are two's complemented using this technique.

**EXAMPLE 1–5**

```
+8 = 00001000
     11110111 (one's complement)
   +        1
     --------
-8 = 11111000 (two's complement)
```

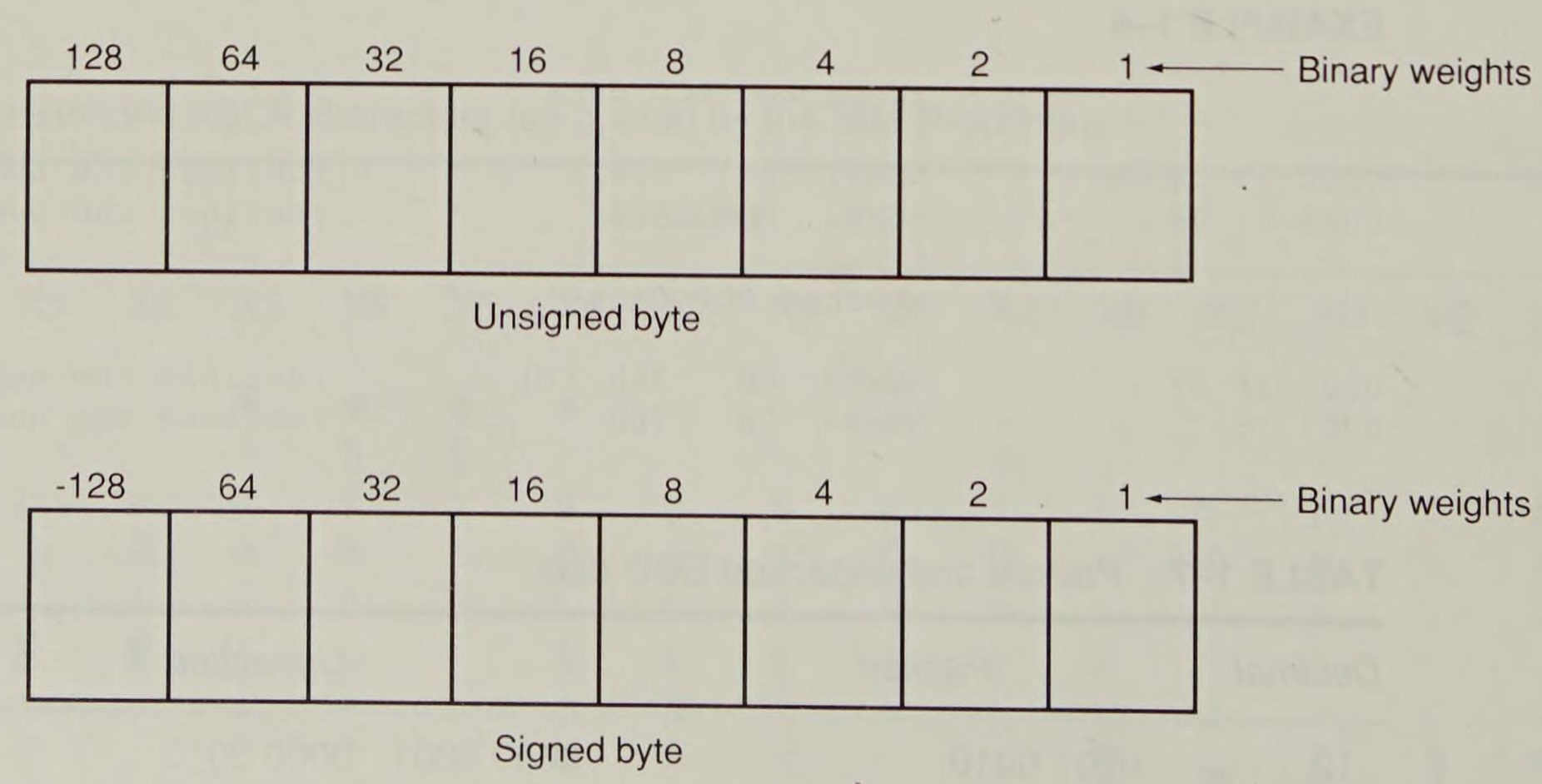



**FIGURE 1–10** The unsigned and signed bytes illustrating the weights of each binary–bit position.

Another, and probably simpler, technique for two's complementing a number starts with the right-most digit. Start writing down the number from right to left. Write the number exactly as it appears until the first one. Write down the first one, and then invert or complement all remaining ones to its left. Example 1–6 shows this technique with the same number as in Example 1–5.

**EXAMPLE 1–6**

```
+8 = 00001000
          1000 (write number to first 1)
     1111      (invert the remaining bits)
-8 = 11111000
```

To store 8-bit data in memory using the assembler program, we use the DB directive as in prior examples. Example 1–7 lists many forms of 8-bit numbers stored in memory using the assembler program.

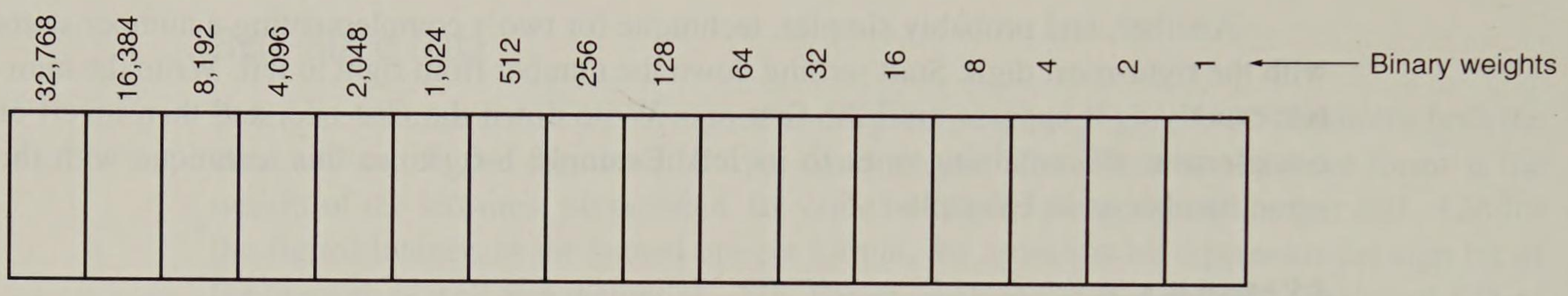
**EXAMPLE 1–7**

```
                        ;Unsigned byte-sized data
                        ;
0000 FE                 DATA1  DB   254             ;define 254 decimal
0001 87                 DATA2  DB   87H             ;define 87 hexadecimal
0002 47                 DATA3  DB   71              ;define 71 decimal
                        ;
                        ;Signed byte-sized data
                        ;
0003 9C                 DATA4  DB   -100            ;define a -100 decimal
0004 64                 DATA5  DB   +100            ;define a +100 decimal
0005 FF                 DATA6  DB   -1              ;define a -1 decimal
```

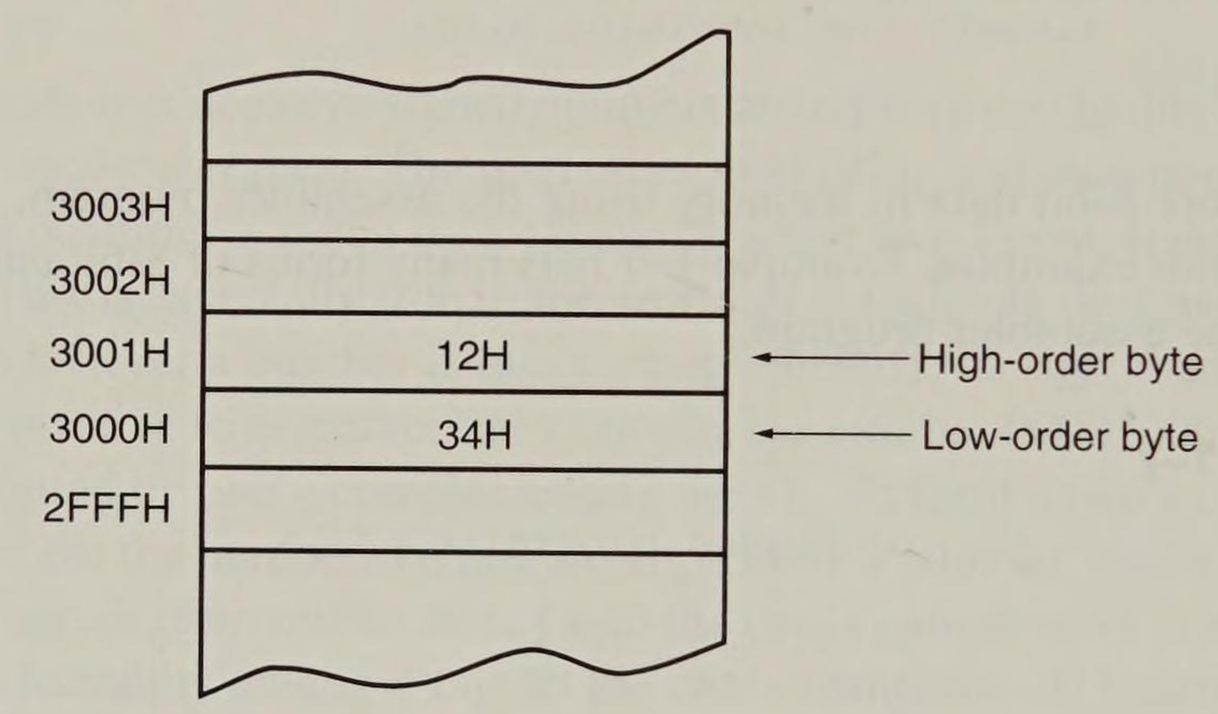
## Word-Sized Data

A *word* (16 bits) is formed with two bytes of data. The least significant byte is always stored in the lowest-numbered memory location, and the most significant byte in the highest. This method of storing a number is called the *little endian* format. An alternate method, not used with the Intel family of microprocessors, is called big endian format. With the *big endian* format numbers are stored with the lowest location containing the most significant data. The big endian format is used with the Motorola family of microprocessors. Figure 1–11(a) shows the weights of each bit position in a word of data, and Figure 1–11(b) shows how the number 1234H appears when stored in the memory location 3000H and 3001H. The only difference between a signed and an unsigned word is the left-most bit position. In the unsigned form, the left-most bit is unsigned; in the signed form, its weight is a –32,768. As with byte-sized signed data, the signed word is in two's complement form when representing a negative number. Also notice that the low-order byte is stored in the lowest-numbered memory location (3000H) and the high-order byte is stored in the highest-numbered location (3001H).

Example 1–8 shows several signed and unsigned word-sized data stored in memory using the assembler program. Notice that the *define word(s)* directive or DW causes the assembler to store words in the memory instead of bytes as in prior examples. Also notice that the word data is displayed by the assembler in the same form as entered. For example,




(a) Unsigned word




(b) The contents of memory location 3000H and 3001H are the word 1234H.

**FIGURE 1–11** The storage format for a 16-bit word in a (a) register and (b) in two bytes of memory.

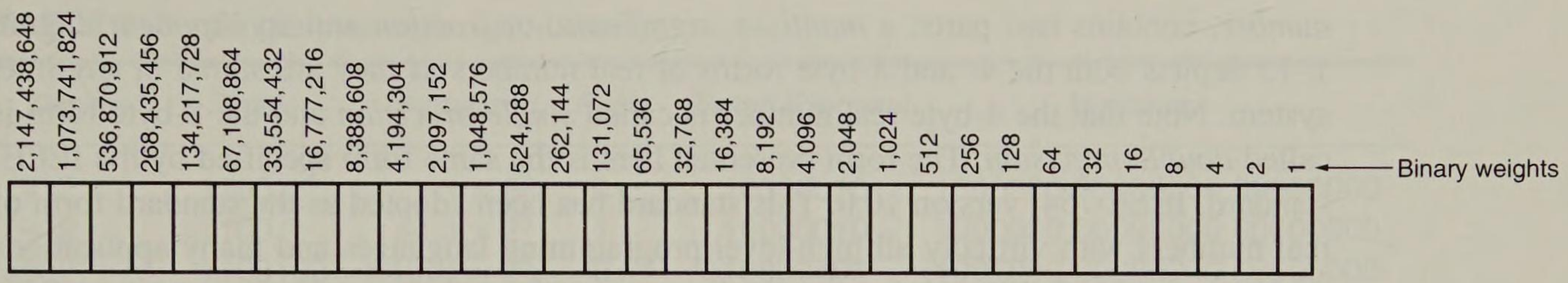
a 1000H is displayed by the assembler as a 1000. This is for our convenience, because the number is actually stored in the memory as a 00 10 in two consecutive memory bytes.

**EXAMPLE 1–8**

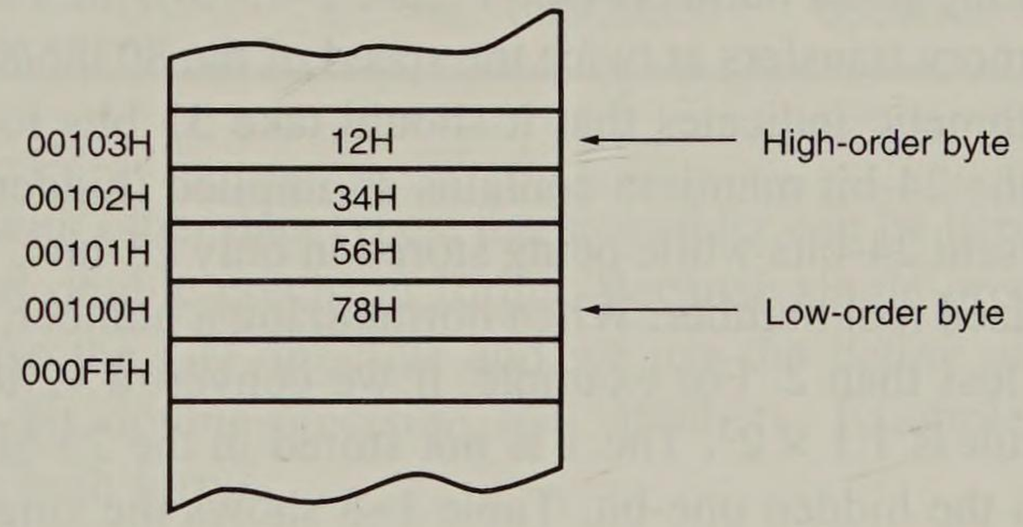
```
                        ;Unsigned word-sized data
                        ;
0000 09F0               DATA1  DW   2544            ;define 2544 decimal
0002 87AC               DATA2  DW   87ACH           ;define 87AC hexadecimal
0004 02C6               DATA3  DW   710             ;define 710 decimal
                        ;
                        ;Signed word-sized data
                        ;
0006 CBA8               DATA4  DW   -13400          ;define a -13,400 decimal
0008 00C6               DATA5  DW   +198            ;define a +198 decimal
000A FFFF               DATA6  DW   -1              ;define a -1 decimal
```

## Doubleword-Sized Data

*Doubleword-sized* data requires four bytes of memory because it is a 32-bit number. Doubleword data appears as a product after a multiplication and also as a dividend before a division. In the 80386/80486 and Pentium microprocessors memory and registers are 32-bits in width. Figure 1–12 shows the form used to store doublewords in the memory and the binary weights of each bit position.




(a) Unsigned doubleword




(b) The contents of memory location 00100H –00103H are the doubleword 12345678H.

**FIGURE 1–12** The storage format for a 32-bit doubleword in a (a) register and (b) in four bytes of memory.

When a doubleword is stored in memory, its least significant byte is stored in the lowest-numbered memory location and its most significant byte is stored in the highest-numbered memory location, using the little endian format. Recall this is also true for word-sized data. For example, a 12345678H that is stored in memory location 00100H–00103H is stored with the 78H in memory location 00100H, the 56H in location 00101H, the 34H in location 00102H, and the 12H in location 00103H.

To define doubleword-sized data, we use the assembler directive *define doubleword(s)* of DD. Example 1–9 shows both signed and unsigned numbers stored in memory using the DD directive.

**EXAMPLE 1–9**

```
                        ;Unsigned doubleword-sized data
                        ;
0000 0003E1C0           DATA1  DD   254400          ;define 254,400 decimal
0004 87AC1234           DATA2  DD   87AC1234H       ;define 87AC1234 hexadecimal
0008 00000046           DATA3  DD   70              ;define 70 decimal
                        ;
                        ;Signed doubleword-sized data
                        ;
000C FFEB8058           DATA4  DD   -1343400        ;define a -1,343,400 decimal
0010 000000C6           DATA5  DD   +198            ;define a +198 decimal
0014 FFFFFFFF           DATA6  DD   -1              ;define a -1 decimal
```

## Real Numbers

Because many high-level languages use the 8086–80486 and Pentium microprocessors, real numbers are often encountered. A real number, or as it is often called, a *floating-point*

*number*, contains two parts: a *mantissa*, *significant*, or *fraction* and an *exponent*. Figure 1–13 depicts both the 4- and 8-byte forms of real numbers as they are stored in any Intel system. Note that the 4-byte real number is called *single-precision* and the 8-byte form is called *double-precision*. The form presented here is the same form specified by the IEEE* standard, IEEE-754, version 10.0. This standard has been adopted as the standard form of real numbers with virtually all high-level programming languages and many applications packages. Figure 1–13(a) shows the single-precision form that contains a sign-bit, an 8-bit exponent, and a 24-bit fraction (mantissa). Note that because applications often require double-precision floating-point numbers (see Figure 1–13(b)), the Pentium with its 64-bit data bus performs memory transfers at twice the speed of the 80386/80486 microprocessors.

Simple arithmetic indicates that it should take 33 bits to store all three pieces of data. Not true—the 24-bit mantissa contains an implied (hidden) one-bit that allows the mantissa to represent 24-bits while being stored in only 23-bits. The *hidden bit* is the first bit of the normalized real number. When normalizing a number, it is adjusted so its value is at least 1, but less than 2. For example, if we convert a 12 to binary (1100) and normalize it, the result is $1.1 \times 2^3$. The 1 is not stored in the 23-bit mantissa portion of the number. The 1 is the hidden one-bit. Table 1–8 shows the single-precision form of this number and others.

The exponent is stored as a biased exponent. With the single-precision form of the real number, the bias is 127 (7FH) and with the double-precision form it is 1023 (3FFH). The *bias* adds to the exponent before it is stored into the exponent portion of the floating-point number. In the previous example, there is an exponent of $2^3$, represented as a biased exponent of 127 + 3 or 130 (82H) in the short form or as 1026 (402H) in the long form.

There are two exceptions to the rules for floating-point numbers. The number 0.0 is stored as all zeros. The number infinity is stored as all ones in the exponent and all zeros in the mantissa. The sign-bit indicates a $\pm$ 0.0 or a $\pm \infty$.

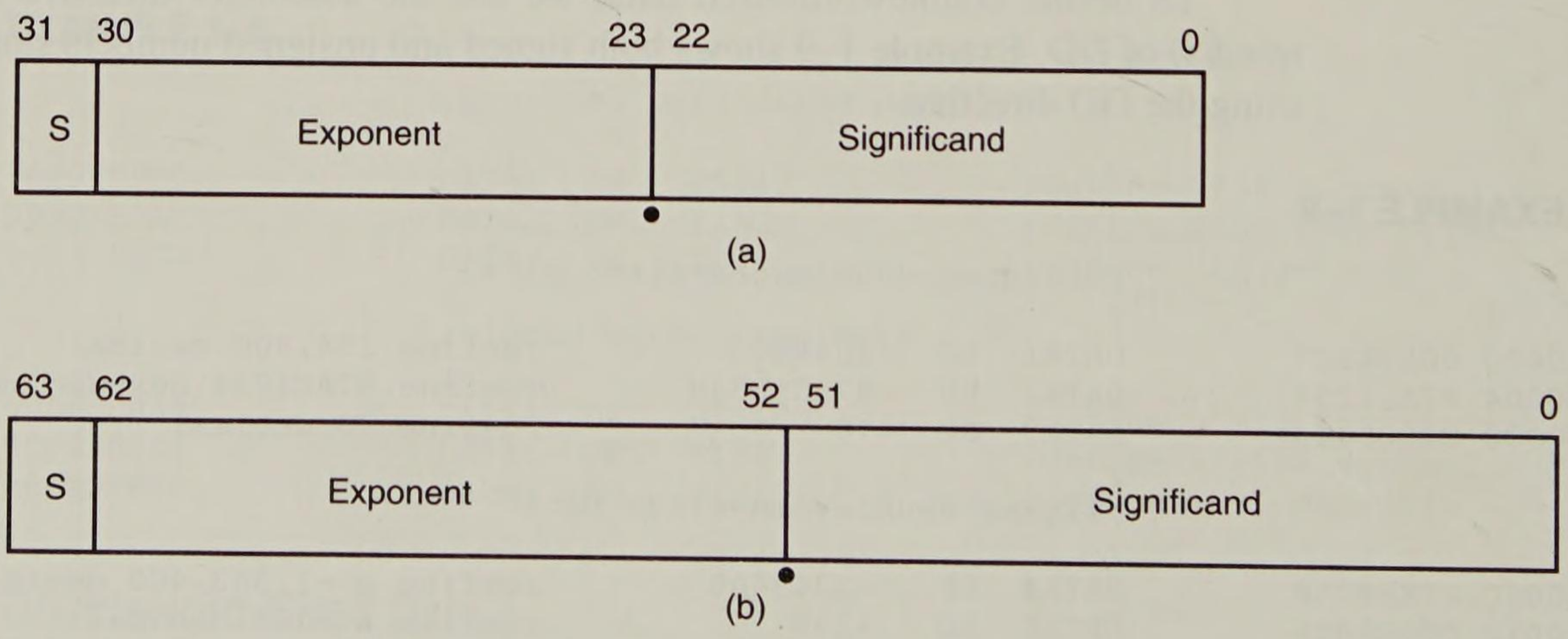



**FIGURE 1–13** The floating-point numbers (a) single-precision using a bias of 7FH, and (b) double-precision using a bias of 3FFH.

*IEEE is the Institute of Electrical and Electronic Engineers.

**TABLE 1–8** Single-precision real numbers

| *Decimal* | *Binary* | *Normalized* | *Sign* | *Biased Exponent* | *Mantissa* |
|---|---|---|---|---|---|
| +12 | 1100 | $1.1 \times 2^3$ | 0 | 10000010 | 1000000 00000000 00000000 |
| –12 | 1100 | $-1.1 \times 2^3$ | 1 | 10000010 | 1000000 00000000 00000000 |
| +100 | 1100100 | $1.1001 \times 2^6$ | 0 | 10000101 | 1001000 00000000 00000000 |
| –1.75 | 1.11 | $-1.11 \times 2^0$ | 1 | 01111111 | 1100000 00000000 00000000 |
| +0.25 | .01 | $1.0 \times 2^{-2}$ | 0 | 01111101 | 0000000 00000000 00000000 |
| +0.0 | 0 | 0 | 0 | 00000000 | 0000000 00000000 00000000 |

As with other data types, the assembler can be used to define real numbers in both single- and double-precision forms. Because single-precision numbers are 32-bit numbers we use the DD directive and we use the *define quadwords(s)* or DQ directive to define 64-bit double-precision real numbers. Example 1–10 shows several numbers defined in both forms.

**EXAMPLE 1–10**

```
                        ;Single-precision real numbers
                        ;
0000 437E6666           DATA1  DD   254.400          ;define 254.400 decimal
0004 C3480000           DATA2  DD   -2.0E2           ;define 200 decimal
0008 428C0000           DATA3  DD   70.0             ;define 70 decimal
                        ;
                        ;Double-precision real numbers
                        ;
000C                    DATA4  DQ   -0.00013434      ;define a -0.00013434 decimal
       BF219BB3D0269E7A
0014                    DATA5  DQ   198.95           ;define a +198.95 decimal
       4068DE6666666666
001C                    DATA6  DQ   -1.0             ;define a -1 decimal
       BFF0000000000000
```

## 1–4 SUMMARY

1. The mechanical computer age began with the invention of the abacus in 500 B.C. This first mechanical calculator remained unchanged until 1642, when Blaise Pascal improved it. An early mechanical computer system was the analytical engine developed by Charles Babbage in 1823. Unfortunately, this machine never functioned because it was impossible to obtain properly machined parts.
2. The first electronic calculating machine was developed during World War II by Konrad Zuse, an early pioneer of digital electronics. His computer, the Z3, was used in aircraft and missile design for the German war effort.
3. The first electronic computer, which used vacuum tubes, was placed into operation in 1943 to break secret German military codes. This first electronic computer system, the Colossus, was invented by Alan Turing. Its program could not be changed.

4. The first general-purpose, programmable electronic computer system was developed in 1946 at the University of Pennsylvania. This first modern computer was called the ENIAC (Electronics Numerical Integrator and Calculator).
5. The world's first microprocessor, the Intel 4004, a 4-bit microprocessor—a programmable controller on a chip—was meager by today's standards. It addressed a mere 4,096 four-bit memory locations. Its instruction set contained only 45 different instructions.
6. Microprocessors that are common today include the 8086/8088, which are the first 16-bit microprocessors. Following these early 16-bit machines were the 80286, 80386, 80486, and Pentium microprocessor. Each newer version increased speed and improved performance.
7. Microprocessor-based personal computers contain memory systems that include three main areas: TPA (transient program area), system area, and extended memory. The TPA holds application programs, the operating system, and drivers. The system area contains memory used for video display cards, disk drives, and the BIOS ROM. The extended memory area is only available to the 80286–80486 and Pentium microprocessor in an AT-style personal computer system.
8. The 8086/8088 address 1M bytes of memory from location 00000H through location FFFFFH. The 80286 and 80386SX address 16M bytes of memory from location 000000H through FFFFFFH. The 80386SL addresses 32M bytes of memory from location 0000000H through 1FFFFFFH. The 80386DX, 80486, and Pentium address 4G bytes of memory from location 00000000H through FFFFFFFFH.
9. All versions of the 8086–80486 and Pentium microprocessor address 64K bytes of I/O address space. These I/O ports are numbered from 0000H through FFFFH with I/O ports 0000H–04FFH reserved for use by the personal computer system.
10. The operating system in many personal computers is either MSDOS (Microsoft disk operating system) or PCDOS (personal computer disk operating system). The operating system performs the task of operating or controlling the computer system along with its I/O devices.
11. The microprocessor is the controlling element in a computer system. The microprocessor performs data transfers, simple arithmetic, and logic operations, and makes simple decisions. The microprocessor executes programs stored in the memory system to perform complex operations in short periods of time.
12. All computer systems contain three buses to control memory and I/O. The address bus is used to request a memory location or I/O device. The data bus transfers data between the microprocessor and its memory and I/O spaces. The control bus controls the memory and I/O and requests reading or writing of data. Control is accomplished with $\overline{\text{IORC}}$ (I/O read control), $\overline{\text{IOWC}}$ (I/O write control), $\overline{\text{MRDC}}$ (memory read control), and $\overline{\text{MWTC}}$ (memory write control).
13. The ASCII code is used to store alphabetic or numeric data. The ASCII code is a 7-bit code and can have an eighth bit used to extend the character set from 128 codes to 256 codes.
14. Binary-coded decimal (BCD) data are sometimes used in a computer system to store decimal data. This data is stored in either packed (2 digits per byte) or unpacked (one digit per byte) form.

15. Binary data are stored as a byte (8 bits), word (16 bits), or doubleword (32 bits) in a computer system. This data may be unsigned or signed. Signed negative data are always stored in the two's complement form. Data that are wider than 8 bits are always stored using the little endian format.
16. Floating-point data are used in a computer system to store whole, mixed, and fractional numbers. A floating-point number is composed of a sign, a mantissa, and an exponent.
17. We use the assembler directive DB to define bytes, DW to define words, DD to define doublewords and DQ to define quadwords.

## 1–5 QUESTIONS AND PROBLEMS

1. Who developed the analytical engine?
2. The 1890 census used a new device called a punch card. Who developed the punch card?
3. Who was the founder of the IBM Corporation?
4. Who developed the first electronic calculator?
5. The first truly electronic computer system was developed for what purpose?
6. The first general-purpose programmable computer was called the _____.
7. The world's first microprocessor was developed in 1971 by _____.
8. Which 8-bit microprocessor ushered in the age of the microprocessor?
9. The 8085 microprocessor, introduced in 1977, has sold _____ copies.
10. Which Intel microprocessor was the first to address 1M bytes of memory?
11. The 80386SL addresses _____ bytes of memory.
12. How much memory is available to the 80486 microprocessor?
13. When did Intel introduce the Pentium microprocessor?
14. What is the meaning of the acronym MIPs?
15. What is the meaning of the acronym CISC?
16. A binary bit stores a _____ or a _____.
17. A computer K is equal to _____ bytes.
18. A computer M is equal to _____ K bytes.
19. A computer G is equal to _____ M bytes.
20. How many typewritten pages of information are stored in a 4G byte memory system?
21. The first 1M byte of memory in a computer system contains a _____ and a _____ area.
22. How much memory is found in the transient program area?
23. How much memory is found in the systems area?
24. The 8086 microprocessor addresses _____ bytes of memory.
25. The 80286 microprocessor addresses _____ bytes of memory.
26. Which microprocessors address 4G bytes of memory?
27. Memory above the first 1M bytes is called _____ memory.
28. What is the BIOS?
29. What is DOS?
30. What is the difference between an XT and an AT computer system?
31. What is the VESA local bus?

32. The ISA bus holds _____-bit interface cards.
33. What is the XMS?
34. What is the EMS?
35. A driver is stored in the _____ area.
36. What is a TSR?
37. How is a TSR often accessed?
38. What is the purpose of the CONFIG.SYS file?
39. What is the purpose of the AUTOEXEC.BAT file?
40. The COMMAND.COM program processes what information?
41. The personal computer system addresses _____ bytes of I/O space.
42. Where is the high memory located in a personal computer?
43. The DEVICE or DEVICEHIGH statement is found in what file?
44. Where are the upper memory blocks used by MSDOS version 5.0 or 6.0?
45. Where is the video BIOS?
46. Draw the block diagram of a computer system.
47. What is the purpose of the microprocessor in a microprocessor-based computer system?
48. List the three buses found in all computer systems.
49. Which bus transfers the memory address to the I/O device or to the memory device?
50. Which control signal causes the memory to perform a read operation?
51. What is the purpose of the $\overline{\text{IORC}}$ signal?
52. If the $\overline{\text{MRDC}}$ signal is a logic 0, which operation is performed by the microprocessor?
53. Define *byte*, *word*, and *doubleword*.
54. Convert the following words into ASCII-coded character strings.
    (a) FROG
    (b) Arc
    (c) Water
    (d) Well
55. What is the difference between an extended ASCII character and a standard ASCII character?
56. Use an assembler directive to store the ASCII-character string 'What time is it?' in the memory.
57. Convert the following decimal numbers into 8-bit signed binary numbers.
    (a) +32
    (b) –12
    (c) +100
    (d) –92
58. Convert the following decimal numbers into signed binary words.
    (a) +1,000
    (b) –120
    (c) +800
    (d) –3,212
59. Use an assembler directive to store –34 into the memory.
60. Show how the following 16-bit hexadecimal numbers are stored in the memory system.
    (a) 1234H
    (b) A122H
    (c) B100H

61 What is the difference between the *big endian* and *little endian* formats for storing numbers that are larger than 8 bits in width?

62. Use an assembler directive to store a 123A hexadecimal in the memory.

63. Convert the following decimal numbers into both packed and unpacked BCD forms.
    (a) 102
    (b) 44
    (c) 301
    (d) 1,000

64. Convert the following binary numbers into signed decimal numbers.
    (a) 10000000
    (b) 00110011
    (c) 10010010
    (d) 10001001

65. Convert the following BCD numbers (assume that these are packed numbers) into decimal numbers.
    (a) 10001001
    (b) 00001001
    (c) 00110010
    (d) 00000001

66. Convert the following decimal numbers into single-precision floating-point numbers.
    (a) +1.5
    (b) –10.625
    (c) +100.25
    (d) –1,200

67. Convert the following single-precision floating-point numbers into decimal.
    (a) 0 10000000 11000000000000000000000
    (b) 1 01111111 00000000000000000000000
    (c) 0 10000010 00100000000000000000000
    (d) 1 01111110 11000000000000000000000

# CHAPTER 2

# The Microprocessor and Its Architecture

## INTRODUCTION

This chapter presents the microprocessor as a programmable device by first looking at its internal programming model and then at how it addresses its memory space. The architecture of the microprocessors is presented as are the ways that various microprocessors address the memory system.

The addressing modes for the Intel family of microprocessors are described for both the real and protected modes of operation. Real mode memory exists at locations 00000H–FFFFFH, the first 1M byte of the memory system, and is present on all versions of the microprocessor. Protected mode memory exists at any location in the entire memory system, but is only available to the 80286–80486 and Pentium and not the earlier 8086 or 8088 microprocessors. Protected mode memory for the 80286 contains 16M bytes and for the 80386, 80486, and Pentium contains 4G bytes.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Describe the function and purpose of each program-visible register in the 8086–80486 and Pentium microprocessors.
2. Detail the flag register and the purpose of each flag bit.
3. Describe how memory is accessed using real mode memory addressing techniques.
4. Describe how memory is accessed using protected mode memory addressing techniques.
5. Describe the program-invisible registers found within the 80286, 80386, 80486, and Pentium microprocessors.

# 2–1 80386, 80486, AND PENTIUM INTERNAL ARCHITECTURES

Before a program is written or any instruction investigated, the internal configuration of the microprocessor must be known. This section of the chapter details the program-visible internal architecture of the 8086–80486 and Pentium microprocessors. Also detailed are the function and purpose of each of these internal registers.

## The Programming Model

The programming model of the 8086–80486 and Pentium microprocessor is considered *program visible* because its registers are used during programming and are specified by the instructions. Other registers, detailed later in this chapter, are considered *program invisible* because they are not normally used during applications programming, but may be used during system programming. Only the 80286–80486 and Pentium contain the program-invisible registers.

Figure 2–1 illustrates the programming model of the 8086–80486 and Pentium microprocessors. The earlier 8086, 8088, and 80286 contain 16-bit internal architectures, a subset of the registers shown in Figure 2–1. The 80386, 80486, and Pentium microprocessors contain full 32-bit internal architectures. The shaded areas in this illustration represent registers that are not found in the 8086, 8088, or 80286 microprocessors and are enhancements provided on the 80386, 80486, and Pentium microprocessors.

The programming model contains 8-, 16-, and 32-bit registers. The *8-bit registers* are AH, AL, BH, BL, CH, CL, DH, and DL and are referred to when an instruction is formed as these two-letter designations. For example, an ADD AL,AH instruction adds AH to AL. The *16-bit registers* are AX, BX, CX, DX, SP, BP, DI, SI, IP, FLAGS, CS, DS, ES, SS, FS, and GS. These registers are also referenced with these two-letter designations. For example, an ADD DX,CX instruction adds CX to DX. The *extended 32-bit registers* are labeled: EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI, EIP, and EFLAGS. These 32-bit extended registers and 16-bit registers FS and GS are only available in the 80386, 80486, and Pentium microprocessors. They are referenced using the register designations FS or GS for the two new 16-bit registers and one of the three-letter designations for the 32-bit registers. For example, an ADD ECX,EBX instruction adds EBX to ECX. Some registers are general-purpose or multipurpose registers, while some have special purposes.

*Multipurpose Registers.* The *multipurpose registers* include EAX, EBX, ECX, EDX, EBP, EDI, and ESI. These registers hold various data sizes (bytes, words, or doublewords) and are used for almost any purpose as dictated by a program.

EAX *(Accumulator)*—used as a 32-bit register (EAX), as a 16-bit register (AX), or as two 8-bit registers AH and AL. Note that if an 8- or 16-bit register is addressed, only that portion of the 32-bit register changes without affecting the remaining bits. The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions. For these instructions, the accumulator has a special purpose, but is generally considered a

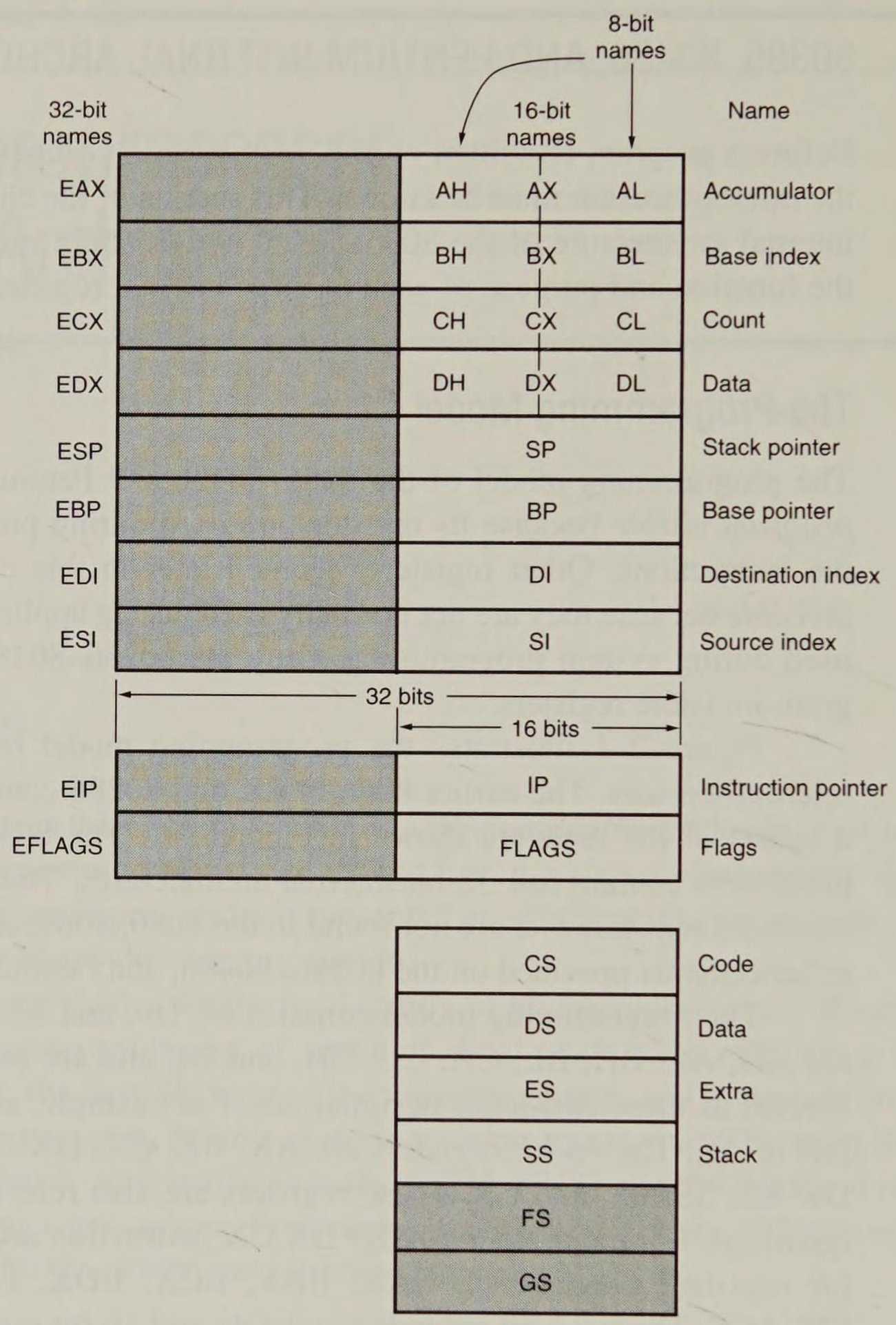



**FIGURE 2–1** The internal program-visible register set of the 8086–80486 and Pentium microprocessors.

Notes:
1. The shaded areas are not available to the 8086, 8088, or 80286 microprocessors.
2. No special names are given to the FS and GS registers.

multipurpose register. In the 80386, 80486, and Pentium microprocessors, the EAX register may also address the memory system.

EBX *(Base Index)*—like EAX, EBX is also addressable as EBX, BX, BH, or BL. The BX register is sometimes used to address memory in all versions of the microprocessor, and in the 80386, 80486, and Pentium, EBX can also be used to address memory data.

ECX *(Count)*— is a general-purpose register that also holds the count for various instructions. In the 80386, 80486, and Pentium, the ECX register can also address memory data. Instructions that use a count are the repeated string instructions (REP), shift, rotate, and LOOP/LOOPD instructions. The shift

and rotate instructions use CL as the count, the repeated string instructions use CX, and LOOP/LOOPD instructions use either CX or ECX as the count.

EDX *(Data)*—a general-purpose register that holds a part of the result from a multiplication or part of the dividend before a division. In the 80386, 80486, and Pentium microprocessor, this register also addresses memory data.

EBP *(Base Pointer)*—can point to a memory location in all versions of the microprocessor for memory data transfers. This register is addressed as either BP or EBP.

EDI *(Destination Index)*— often addresses string destination data for the string instructions. It also functions as either a 32-bit (EDI) or 16-bit (DI) general-purpose register.

ESI *(Source Index)*—used as either ESI or SI. The source index register often addresses source string data for the string instructions. Like EDI, ESI also functions as a general-purpose register. As a 16-bit register it is addressed as SI and as a 32-bit register as ESI.

*Special-Purpose Registers.* The special-purpose registers include: EIP, ESP, EFLAGS, and the segment registers CS, DS, ES, SS, FS, and GS.

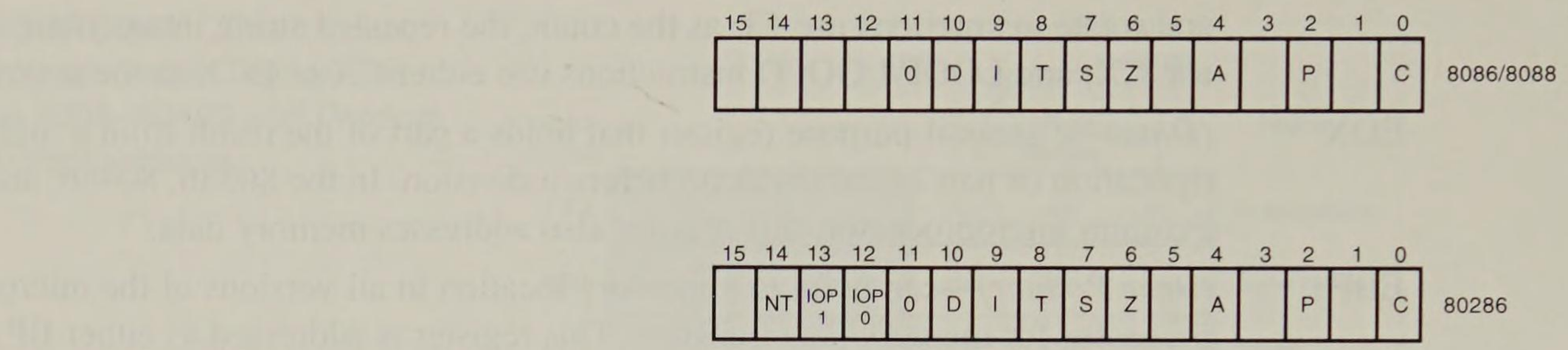
EIP *(Instruction Pointer)*—addresses the next instruction in a section of memory defined as a code segment. This register is IP (16 bits) when the 8086–80486 and Pentium operate in the real mode and EIP (32 bits) when the 80386, 80486, and Pentium operate in the protected mode. Note that the 8086, 8088, and 80286 do contain EIP and only the 80286, 80386, 80486, and Pentium operate in the protected mode. The instruction pointer, which points to the next instruction in a program, is used by the microprocessor to find the next sequential instruction in a program located within the code segment. The instruction pointer can be modified with a jump or a call instruction.

ESP *(Stack Pointer)*—addresses an area of memory called the stack. The stack memory stores data through this pointer and is explained later in the text with instructions that address stack data.

EFLAGS *(Extended Flags)*—indicates the condition of the microprocessor and controls its operation. Figure 2–2 shows the flag registers of all versions of the microprocessor. Note that the flags are upward compatible from the 8086/8088 to the Pentium microprocessor. The 8086–80286 contain a FLAG register (16 bits) and the 80386, 80486, and Pentium contain an EFLAG register (32 bit extended flag register).

The right-most five flag bits and the overflow flag change after many arithmetic and logic instructions execute. Some of the flags are also used to control features found in the microprocessor. Following is a list of each flag bit with a brief description of its function. As instructions are introduced in subsequent chapters, additional detail on the flag bits is provided.

C *(Carry)*—holds the carry after addition or the borrow after subtraction. The carry flag also indicates error conditions in some programs and procedures.




Notes: 1. The blank bits are reserved for future use and must not be defined.
2. The Pentium EFLAG register is illustrated in Chapter 14.

**FIGURE 2–2** The flag registers of all family members except the Pentium. Note how they are all upward compatible.

This is especially true of the DOS function calls detailed in later chapters and Appendix A.

P *(Parity)*—is a logic 0 for odd parity and a logic 1 for even parity. Parity is a count of ones in a number expressed as even or odd. For example, if a number contains 3 binary one bits, it has odd parity. If a number contains zero one bits it is considered to have even parity.

A *(Auxiliary Carry)*—holds the carry (half-carry) after addition or the borrow after subtraction between bits positions 3 and 4 of the result. This highly specialized flag bit is tested by the DAA and DAS instructions to adjust the value of AL after a BCD addition or subtraction. Otherwise, the A flag bit is not used by the microprocessor or any other instructions.

Z *(Zero)*—shows that the result of an arithmetic or logic operation is zero. If Z = 1, the result is zero, and if Z = 0, the result is not zero.

S (*Sign*)—holds the arithmetic sign of the result after an arithmetic or logic instruction executes. If S = 1, the sign is set or negative, and if S = 0, the sign is cleared or positive.

T *(Trap)*—enables trapping through an on-chip debugging feature. More detail of this debugging feature is provided later in the text with a discussion of the debug registers and debugging features provided by the microprocessor. (We *debug* a program to find an error or *bug*.)

I *(Interrupt)*—controls the operation of the INTR (interrupt request) input pin. If I = 1, the INTR pin is enabled, and if I = 0, the INTR pin is disabled. The

state of the I flag bit is controlled by the STI (set I flag) and CLI (clear I flag) instructions.

D *(Direction)*—selects either the increment or decrement mode for the DI and/or SI registers during string instructions. If D = 1, the registers are automatically decremented and if D = 0, the registers are automatically incremented. The D flag is set with the STD (set direction) and cleared with the CLD (clear direction) instructions.

O *(Overflow)*—occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine. For example, if a 7FH (+127) is added—using an 8-bit addition—to a 01H (+1), the result is 80H (–128). This result represents an overflow condition indicated by the overflow flag for signed addition. For unsigned operations, we merely ignore the overflow flag.

IOPL *(Input/Output Privilege Level)*—used in protected mode operation to select the privilege level for I/O devices. If the current privilege level is higher or more trusted than the IOPL, then I/O executes without hindrance. If the IOPL is lower than the current privilege level, an interrupt occurs causing execution to suspend. Note that an IOPL of 00 is the highest or most trusted and an IOPL of 11 is the lowest or least trusted.

NT *(Nested Task)*—indicates that the current task is nested within another task in protected mode operation. This flag is set when the task is nested by software.

RF *(Resume)*—used with debugging to control the resumption of execution after the next instruction.

VM *(Virtual Mode)*—selects virtual mode operation in a protected mode system. A virtual mode system allows multiple DOS memory partitions that are 1M byte in length to coexist in the memory system. This allows the system program to execute multiple DOS programs.

AC *(Alignment Check)*—activates if a word or doubleword is addressed on a nonword or nondoubleword boundary. Only the 80486SX microprocessor contains the alignment check bit that is primarily used by its companion numeric coprocessor, the 80487SX, for synchronization.

*Segment Registers.* Additional registers, called segment registers, generate memory addresses when combined with other registers in the microprocessor. There are either 4 or 6 segment registers in various versions of the 8086–80486 and Pentium microprocessors. A segment register functions differently in the real mode when compared to protected mode operation of the microprocessor. Detail on their function in real and protected mode is provided later in this chapter. Following is a list of each segment register along with its function in the system:

CS *(Code)*—the code segment is a section of memory that holds the code (*programs and procedures*) used by the microprocessor. The code segment register defines the starting address of the section of memory holding code. In real mode operation it defines the start of a 64K byte section of memory and

in protected mode it selects a descriptor that describes the starting address and length of a section of memory holding code. The code segment is limited to 64K bytes in length in the 8086–80286 and 4G bytes in the 80386, 80486, and Pentium when these microprocessors operate in the protected mode.

DS *(Data)*—the data segment is a section of memory that contains most data used by a program. Data are accessed in the data segment by an offset address or the contents of other registers that hold the offset address. As with the code segment and other segments, the length is limited to 64K bytes in the 8086–80286 and 4G bytes in the 80386, 80486, and Pentium.

ES *(Extra)*—the extra segment is an additional data segment that is used by some of the string instructions to hold destination data.

SS *(Stack)*—the stack segment defines the area of memory used for the stack. The location of the current entry point in the stack segment is determined by the stack pointer register. The BP register also addresses data within the stack segment.

FS and GS —these supplemental segment registers are available in the 80386, 80486, and Pentium microprocessors to allow two additional memory segments for access by programs.

## 2–2 REAL-MODE MEMORY ADDRESSING

The 80286–80486 and Pentium microprocessors operate in either the real or protected mode. The 8086 and 8088 only operate in the real mode. This section of the text details the operation of the microprocessor in the real mode. Real mode operation allows the microprocessor to only address the first 1M byte of memory space, even if it is the Pentium microprocessor. Both the MSDOS or PCDOS operating systems assume that the microprocessor is operated in the real mode at all times. Real-mode operation allows application software written for the 8086/8088, which only contain 1M byte of memory, to function in the 80286, 80386, 80486, and Pentium microprocessors. The upward compatibility of software is partially responsible for the success of the Intel family of microprocessors. In all cases, each of these microprocessors begins operation in the real mode by default, whenever power is applied or the microprocessor is reset.

### Segments and Offsets

A combination of a segment address and an offset address access a memory location in the real mode. All real-mode memory addresses consist of a segment address plus an offset address. The *segment address*, located within one of the segment registers, defines the beginning address of any 64K-byte memory segment. The *offset address* selects a location within the 64K-byte memory segment. Figure 2–3 shows how the segment plus offset addressing scheme selects a memory location. This illustration shows a memory segment that begins at location 10000H and ends at location 1FFFFH, 64K bytes in length. It also

shows how an offset, sometimes called a *displacement*, of F000H selects location 1F000H in the memory system. Note that the *offset* or displacement is the distance above the start of the segment.

The segment register in Figure 2–3 contains a 1000H yet it addresses a starting segment at location 10000H. In the real mode, each segment register is internally appended with a *0H* on its right-most end. This forms a 20-bit memory address, allowing it to access the start of a segment at any location within the first 1M byte of memory. For example, if a segment register contains a 1200H, it addresses a 64K-byte memory segment beginning at location 12000H. Likewise, if a segment register contains a 1201H, it addresses a memory segment beginning at location 12010H. Because of the internally appended 0H, real-mode segments can only begin at a 16-byte boundary in the memory system. We often call this 16-byte boundary a *paragraph*.

Because a real-mode segment of memory is 64K in length, once the beginning address is known, the ending address is found by adding FFFFH. For example, if a segment register contains 3000H, the first address of the segment is 30000H and the last address is 30000H + FFFFH, or 3FFFFH. Table 2–1 shows several examples of segment register contents and the starting and ending addresses of the memory segments selected by each segment address.

The offset address is added to the start of the segment to address a memory location in the memory segment. For example, if the segment address is 1000H and the offset address is 2000H, the microprocessor addresses memory location 12000H. The segment and offset address is sometimes written as *1000:2000* for a segment address of 1000H with an offset of 2000H.

In the 80286 (with special external circuitry), 80386, 80486, and Pentium, an extra 64K minus 16 bytes of memory is addressable when the segment address is FFFFH and the HIMEM.SYS driver is installed in the system. This area of memory (FFFF0H–10FFEFH)

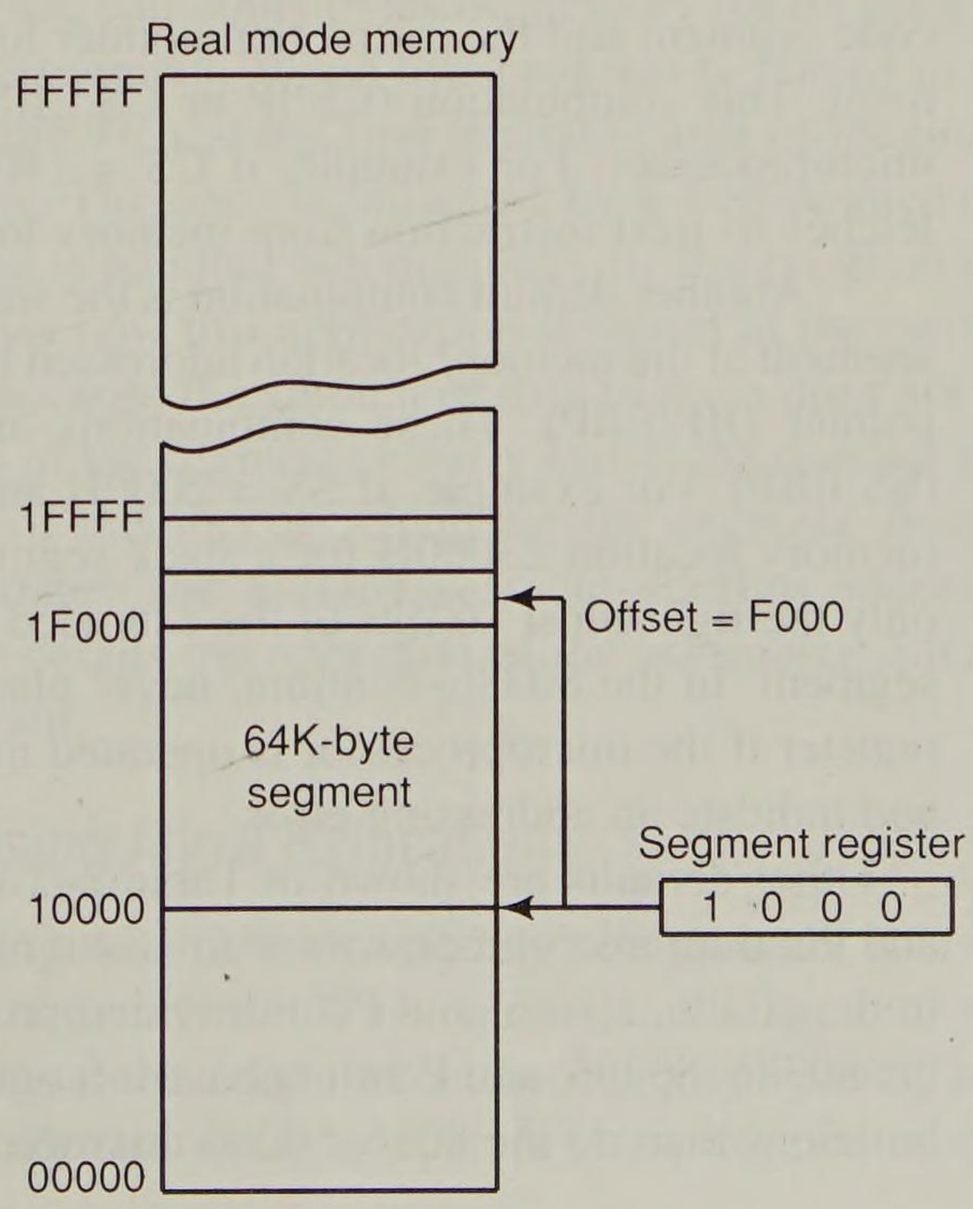



**FIGURE 2–3** The real-mode memory addressing scheme using a segment address plus an offset.

**TABLE 2–1** Example segment addresses

| *Segment Register* | *Starting Address* | *Ending Address* |
|---|---|---|
| 2000H | 20000H | 2FFFFH |
| 2100H | 21000H | 30FFFH |
| AB00H | AB000H | BAFFFH |
| 1234H | 12340H | 2233FH |

is referred to as high memory. When an address is generated using a segment address of FFFFH, the A20 address pin is set when an offset is added. For example, if the segment address if FFFFH and the offset address is 4000H, the machine addresses memory location FFFF0H + 4000H, or 103FF0H.

Some addressing modes combine registers to form an offset address. When this occurs, the offset address may exceed FFFFH in value. For example, the address accessed in a segment whose segment address is 1000H and whose offset address is specified as the sum of F000H plus 3000H will access memory location 12000H. When the F000H and 3000H are added they form a 16-bit (modulo 16) sum of 2000H used as the offset address. Note that the carry of 1 (F000H + 3000H = 12000H) is dropped for this addition to form the offset address of 2000H.

## Default Segment and Offset Registers

The microprocessor has a set of rules that apply to segments whenever memory is addressed. These rules, which apply in either the real or protected mode, define the segment register and offset register combination used by certain addressing modes. For example, the code segment register is always used with the instruction pointer to address the next instruction in a program. This combination is CS:IP or CS:EIP depending upon the microprocessor and mode of operation. The code segment register defines the start of the code segment and the instruction pointer locates the next instruction within the code segment. This combination (CS:IP or CS:EIP) locates the next instruction executed by the microprocessor. For example, if CS = 1400H and IP/EIP = 1200H, the microprocessor fetches its next instruction from memory location 14000H + 1200H or 15200H.

Another default combination is the stack. Stack data are referenced through the stack segment at the memory location addressed by either the stack pointer (SP/ESP) or the base pointer (BP/EBP). These combinations are referred to as SS:SP (SS:ESP) or SS:BP (SS:EBP). For example, if SS = 2000H and BP = 3000H, the microprocessor addresses memory location 23000H for a stack segment memory location. Note that in real mode, only the right-most 16-bits of the extended register address a location within the memory segment. In the 80386–Pentium, never place a number larger than FFFFH into an offset register if the microprocessor is operated in the real mode. This causes the system to halt and indicate an addressing error.

Other defaults are shown in Table 2–2 for addressing memory using the 8086–80486 and Pentium microprocessors with 16-bit registers. Table 2–3 shows the defaults assumed in the 80386, 80486, and Pentium microprocessors when using 32-bit registers. Note that the 80386, 80486, and Pentium have a far greater selection of segment/offset address combinations than do the 8086–80286 microprocessors.

**TABLE 2–2** 8086–80486 and Pentium default 16-bit segment and offset addresses

| *Segment* | *Offset* |
|---|---|
| CS | IP |
| SS | SP or BP |
| DS | BX, DI, SI, or a 16-bit number |
| ES | DI for string instructions |

**TABLE 2–3** 80386, 80486, and Pentium default 32-bit segment and offset addresses

| *Segment* | *Offset* |
|---|---|
| CS | EIP |
| SS | ESP or EBP |
| DS | EAX, EBX, ECX, EDX, EDI, ESI, an 8-bit number, or 32-bit number |
| ES | EDI for string instructions |
| FS | no default |
| GS | no default |

The 8086–80286 allow four memory segments and the 80386, 80486, and Pentium allow six memory segments. Figure 2–4 shows a system that contains four memory segments. Note that a memory segment can touch or even overlap if 64K bytes of memory is not required for a segment. Think of segments as windows that can be moved over any area of memory to access data or code. Also note that a program can have more than four or six segments, but can only access four or six segments at a time.

Suppose an application program requires 1000H bytes of memory for its code, 190H bytes of memory for its data, and 200H bytes of memory for its stack. This application does not require an extra segment. When this program is placed in the memory system by DOS, it is loaded in the TPA at the first available area of memory above the drivers and other TPA programs. This area is indicated by a free-pointer that is maintained by DOS. Program loading is handled automatically by the program loader located within DOS. Figure 2–5 shows how this application is stored in the memory system. The segments show an overlap because the amount of data in them does not require 64K bytes of memory. The side view of the segments clearly shows the overlap and how segments can be slid to any area of memory by changing the segment starting address. Fortunately for us, DOS calculates and assigns segment starting addresses. This is explained in a later chapter that details the operation of the assembler, BIOS, and DOS for an assembly language program.

## Segment and Offset Addressing Allow Relocation

The segment and offset addressing scheme seems unduly complicated. It is complicated, but it also affords an advantage to the system. This complicated scheme of segment plus offset addressing allows programs to be relocated in the memory system. It also allows programs written to function in the real mode to operate in a protected-mode system. A

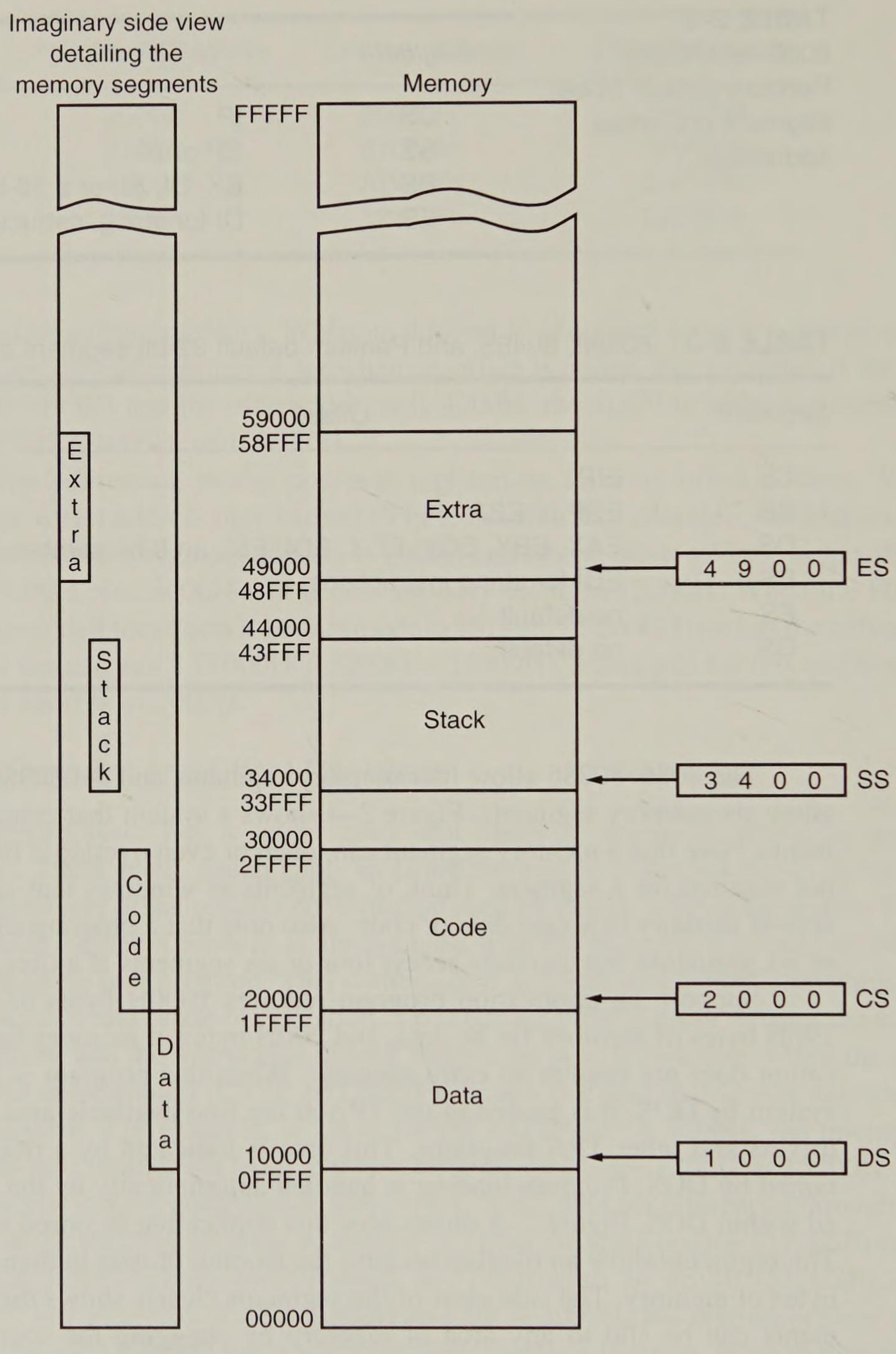



**FIGURE 2–4** An example memory system showing the placement of four memory segments.

*relocatable program* is one that can be placed into any area of memory and executed without change. *Relocatable data* is data that can be placed in any area of memory and used without any change to the program. The segment and offset addressing scheme allows both programs and data to be relocated without changing a thing in a program or data. This is ideal for use in a general-purpose computer system, where not all machines contain the

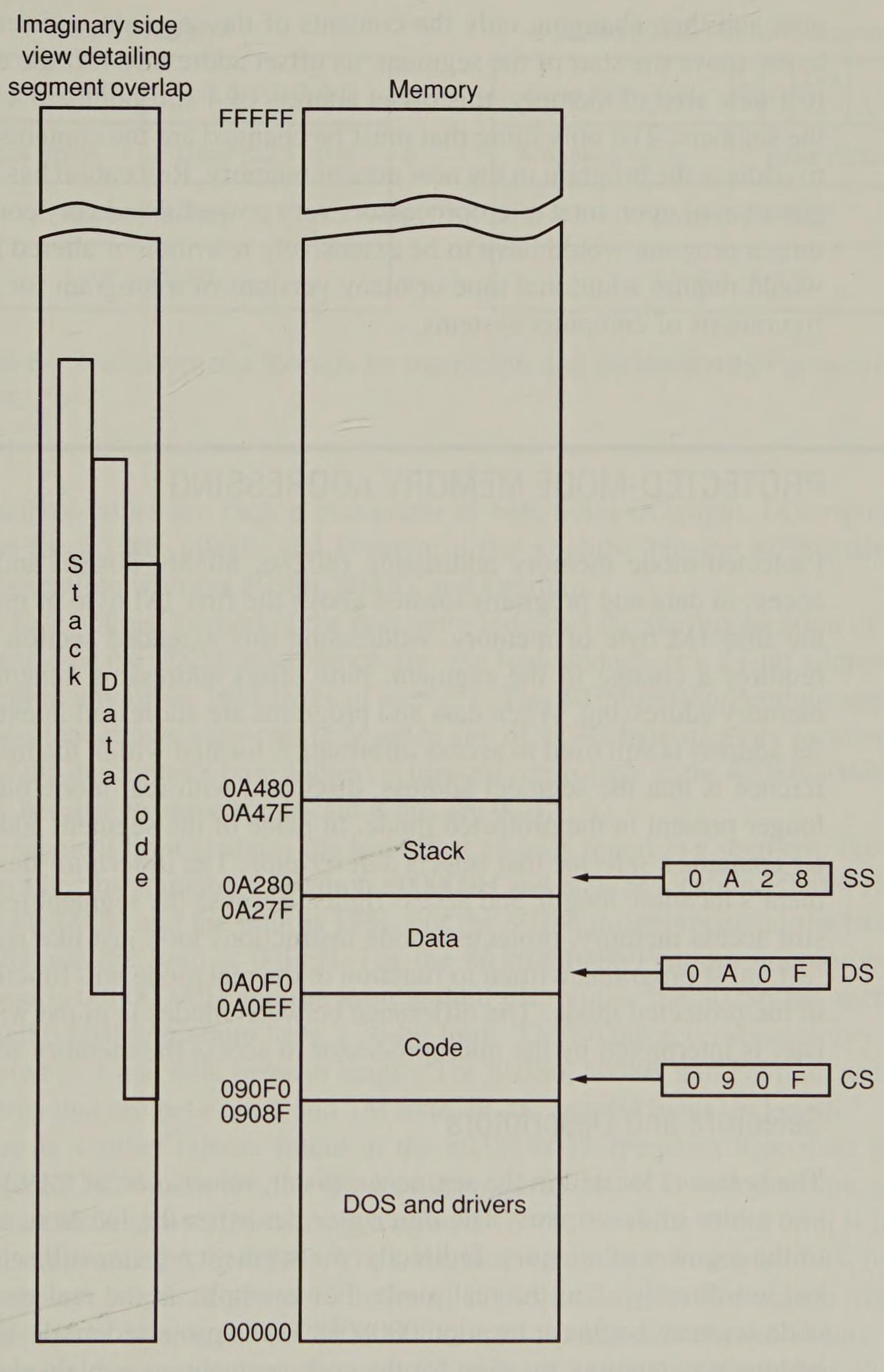



**FIGURE 2–5** An application program containing a code, data, and stack segment loaded into a DOS system memory. Notice how these three segments overlap.

same memory areas. The personal computer memory structure is different from machine to machine, requiring relocatable software and data.

Because memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses. This is accomplished by moving the entire program, as a block, to a new

area and then changing only the contents of the segment registers. If an instruction is 4 bytes above the start of the segment, its offset address is 4. If the entire program is moved to a new area of memory, this offset address of 4 still points to 4 bytes above the start of the segment. The only thing that must be changed are the contents of the segment register to address the program in the new area of memory. Relocation has made the personal computer based upon Intel microprocessors very powerful and very common. Without this feature, a program would have to be extensively rewritten or altered before it is moved. This would require additional time or many versions of a program for the many different configurations of computer systems.

## 2–3 PROTECTED-MODE MEMORY ADDRESSING

Protected-mode memory addressing (80286, 80386, 80486, and Pentium only) allows access to data and programs located above the first 1M byte of memory as well as within the first 1M byte of memory. Addressing this extended section of the memory system requires a change to the segment, plus offset addressing scheme used with real-mode memory addressing. When data and programs are addressed in extended memory, the offset address is still used to access information located within the memory segment. The difference is that the segment address, discussed with real-mode memory addressing, is no longer present in the protected mode. In place of the segment address, the segment register contains a *selector* that selects a *descriptor*. The *descriptor* describes the memory segment's location, length, and access rights. Because the segment register and offset address still access memory, protected-mode instructions look just like real-mode instructions. In fact, most programs written to function in the real mode will function without any changes in the protected mode. The difference between modes is in the way that the segment register is interpreted by the microprocessor to access the memory segment.

### Selectors and Descriptors

The *selector*, located in the segment register, selects one of 8,192 descriptors from one of two tables of descriptors. The *descriptor* describes the location, length, and access rights of the segment of memory. Indirectly, the segment register still selects a memory segment, but not directly as in the real mode. For example, in the real mode, if CS = 0008H, the code segment begins at location 00080H. In the protected mode, this segment number can address any memory location for the code segment, as explained shortly.

There are two descriptor tables used with the segment registers: one contains global descriptors and the other contains local descriptors. The *global descriptors* contain segment definitions that apply to all programs, while the *local descriptors* are usually unique to an application. Each descriptor table contains 8,192 descriptors, so a total of 16,384 descriptors are available to an application at any time. Because the descriptor describes a memory segment, this allows up to 16,384 memory segments to be described for each application.

Figure 2–6 shows the format of a descriptor for the 80286, 80386, 80486, and Pentium microprocessors. Note that each descriptor is 8 bytes in length so the global and

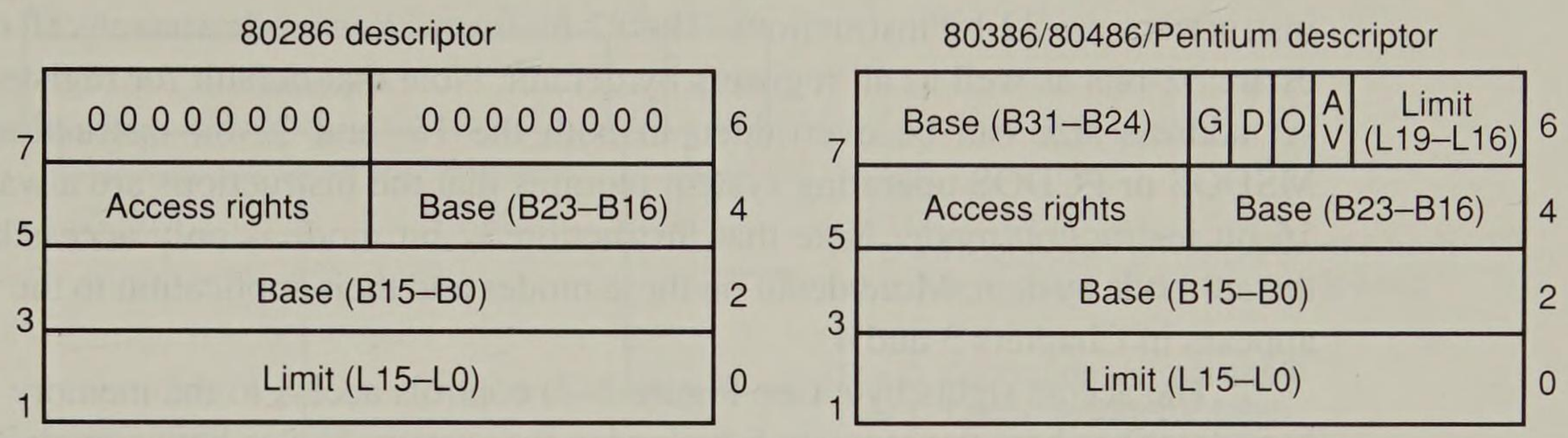



**FIGURE 2–6** The descriptor formats for the 80286 and 80386/80486/Pentium microprocessors.

local descriptor tables are each a maximum of 64K bytes in length. Descriptors for the 80286 and the 80386, 80486, and Pentium differ slightly, but the 80286 descriptor is upward compatible with the 80386, 80486, and Pentium.

The base address portion of the descriptor indicates the starting location of the memory segment. For the 80286 microprocessor, the base address is a 24-bit address, so segments begin at any of its 16M bytes of memory. The 80386/80486/Pentium uses a 32-bit base address that allows segments to begin at any of its 4G byte memory locations. Notice how the 80286 descriptor's base address is upward compatible to the 80386/80486/Pentium descriptor, because its most significant 8 bits are 00H.

The segment limit contains the last offset address found in a segment. For example, if a segment begins at memory location F00000H and ends at location F000FFH, the base address is F00000H and the limit is FFH. For the 80286 microprocessor, the base address is F00000H and the limit is 00FFH. For the 80386/80486/Pentium microprocessor, the base address is 00F00000H and the limit is 000FFH. Notice the 80286 has a 16-bit limit and the 80386/80486/Pentium have a 20-bit limit. The 80286 accesses memory segments that are between 1 and 64K bytes in length. The 80386, 80486, and Pentium access memory segments that are between 1 and 1M byte, or 4K and 4G bytes, in length.

There is another feature found in the 80386/80486/Pentium descriptor that is not found in the 80286 descriptor: the G bit or *granularity* bit. If G = 0, the limit specifies a segment limit of from 1 to 1M bytes in length. If G = 1, the value of the limit is multiplied by 4K bytes (appended with 000H). Thus, the limit can be any multiple of 4K bytes. This allows a segment length of 4K to 4G bytes in steps of 4K bytes. The reason that the segment length is 64K byte in the 80286 is that the offset address is always 16 bits, while the offset address in the protected-mode operation of the 80386/80486/Pentium is 32 bits. This 32-bit offset address allows segment lengths of 4G bytes, and the 16-bit offset address allows segment lengths of 64K bytes.

The AV bit in the 80386/80486/Pentium descriptor is used by some operating systems to indicate that the segment is available (AV = 1) or not available (AV = 0). The D bit indicates how the 80386/80486/Pentium instructions access register and memory data in the protected mode. If D = 0, the 80386/80486/Pentium assumes that the instructions are 16-bit instructions compatible with the 8086–80286 microprocessors. This means that the instructions use 16-bit offset addresses and 16-bit registers by default. This mode is often called the *16-bit instruction mode*. If D = 1, the 80386/80486/Pentium assumes the

instructions are 32-bit instructions. The *32-bit instruction mode* assumes all offset addresses are 32-bits as well as all registers by default. Note that default for register size and offset address size can be overridden in both the 16- and 32-bit instruction modes. The MSDOS or PCDOS operating system requires that the instructions are always used in the 16-bit instruction mode. Note that instruction 32-bit mode is only accessible in the protected-mode system. More detail on these modes and their application to the instruction set appears in Chapters 3 and 4.

The access rights byte (see Figure 2–7) controls access to the memory segment. This byte describes how the segment functions in the system. Notice how complete the control is over the segment. If the segment is a data segment, the direction of growth can be specified. If the segment grows beyond its limit, the microprocessor's program is interrupted. You can even specify if a data segment can be written or is write-protected. The code segment is also controlled in a similar fashion and can have reading inhibited to protect software.

Descriptors are chosen from the descriptor table by the segment register. Figure 2–8 shows how the segment register functions in the protected-mode system. The segment register contains a 13-bit selector field, a table selector bit, and a requested privilege level field. The 13-bit selector chooses one of the 8,192 descriptors from the descriptor table. The TI bit selects either the global descriptor table (TI = 0) or the local descriptor table (TI = 1). The requested privilege level (RPL) requests the access privilege level of a memory segment. The highest privilege level is 00 and the lowest is 11. If the requested privilege level matches or is higher in priority than the privilege level set by the access rights byte, access is granted. For example, if the requested privilege level is 10 and the access rights

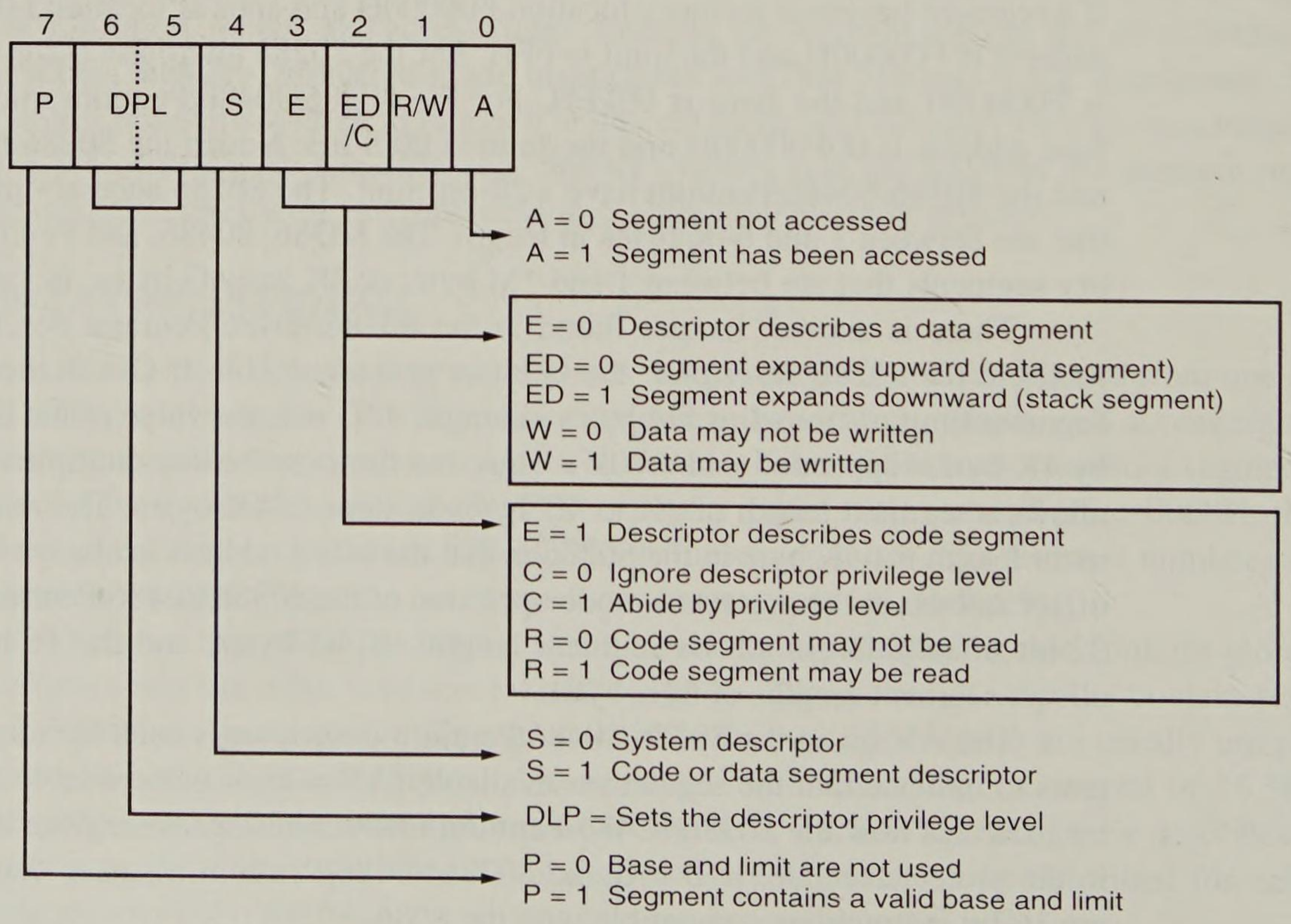



**FIGURE 2–7** The access rights byte for the 80286, 80386, 80486, and Pentium descriptor.

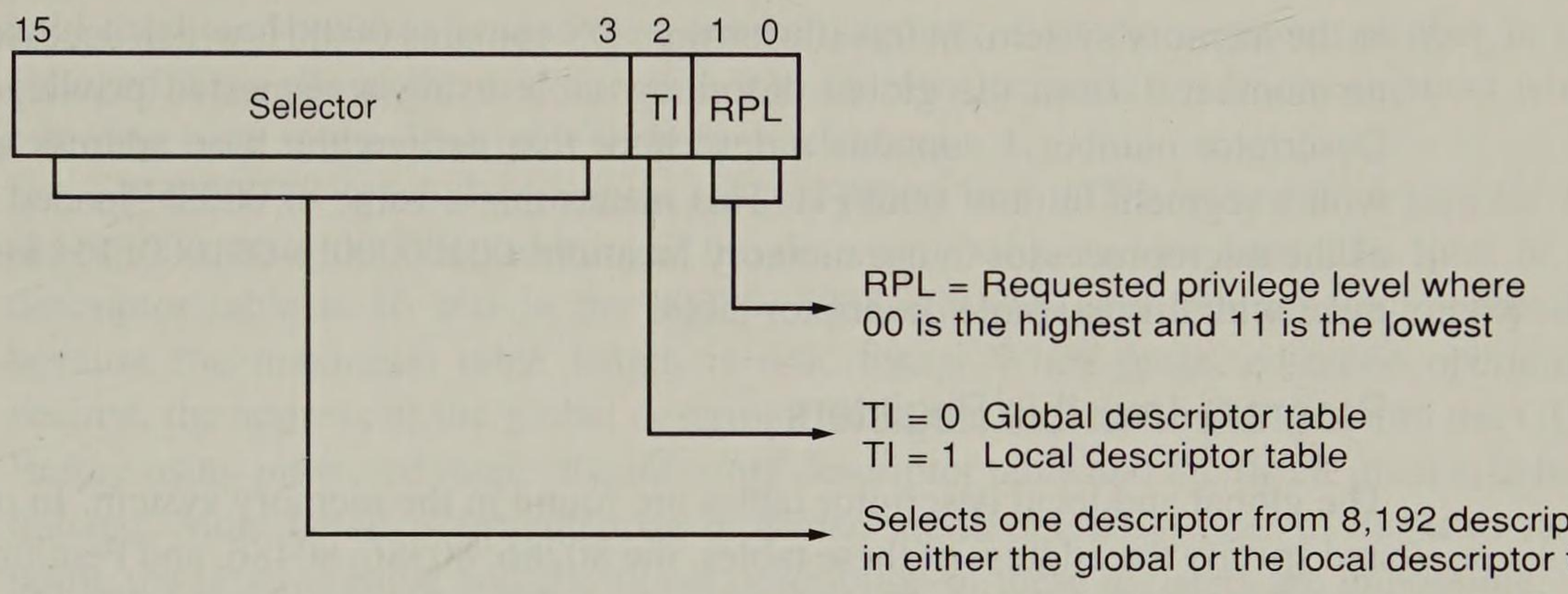



**FIGURE 2–8** The contents of a segment register during protected-mode operation of the 80286, 80386, 80486, and Pentium microprocessors.

byte sets the segment privilege level at 11, access is granted because 10 is higher in priority than privilege level 11. Privilege levels are used in multiuser environments.

Figure 2–9 shows how the segment register, containing a selector, chooses a descriptor from the global descriptor table. The entry in the global descriptor table selects a segment

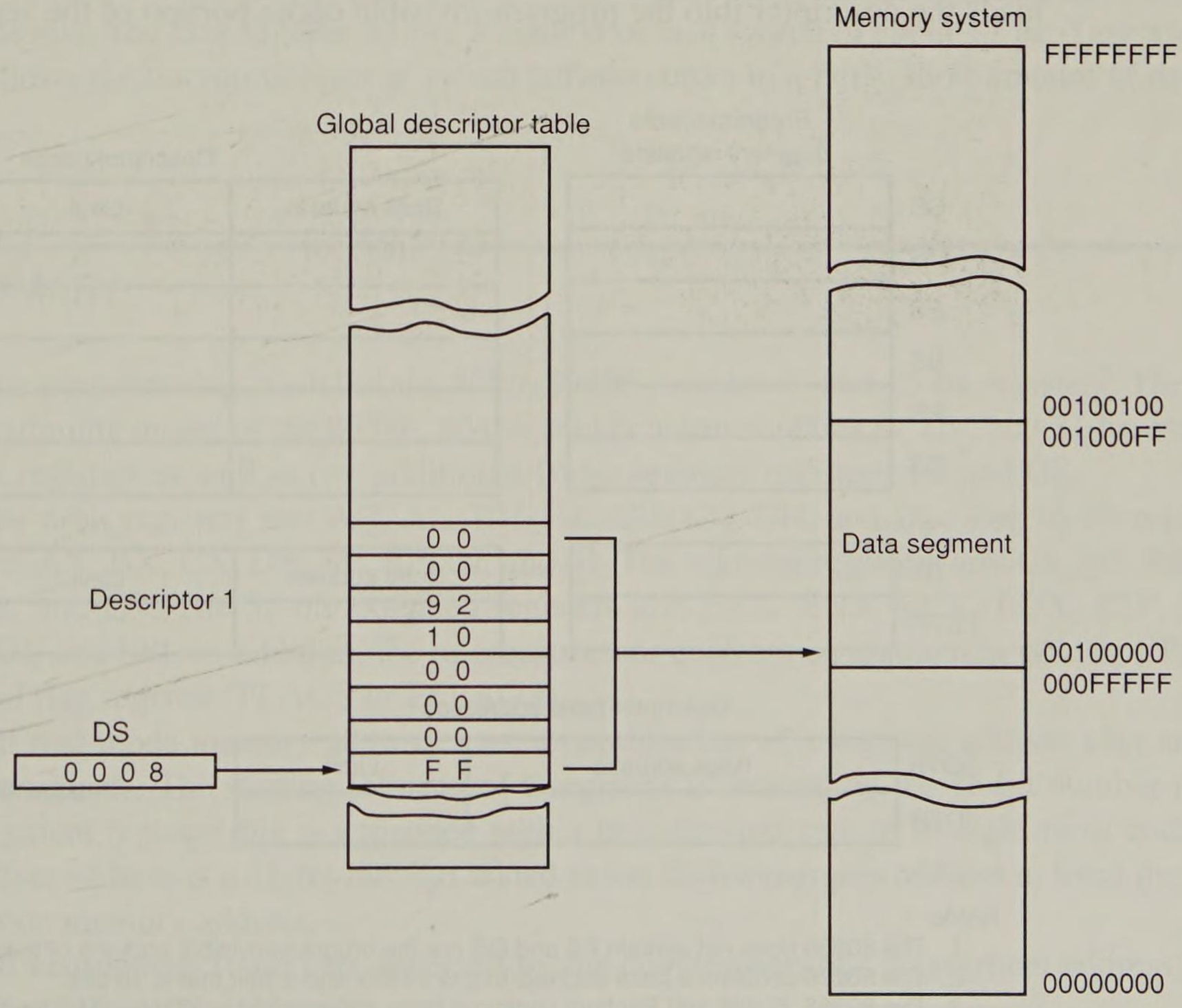


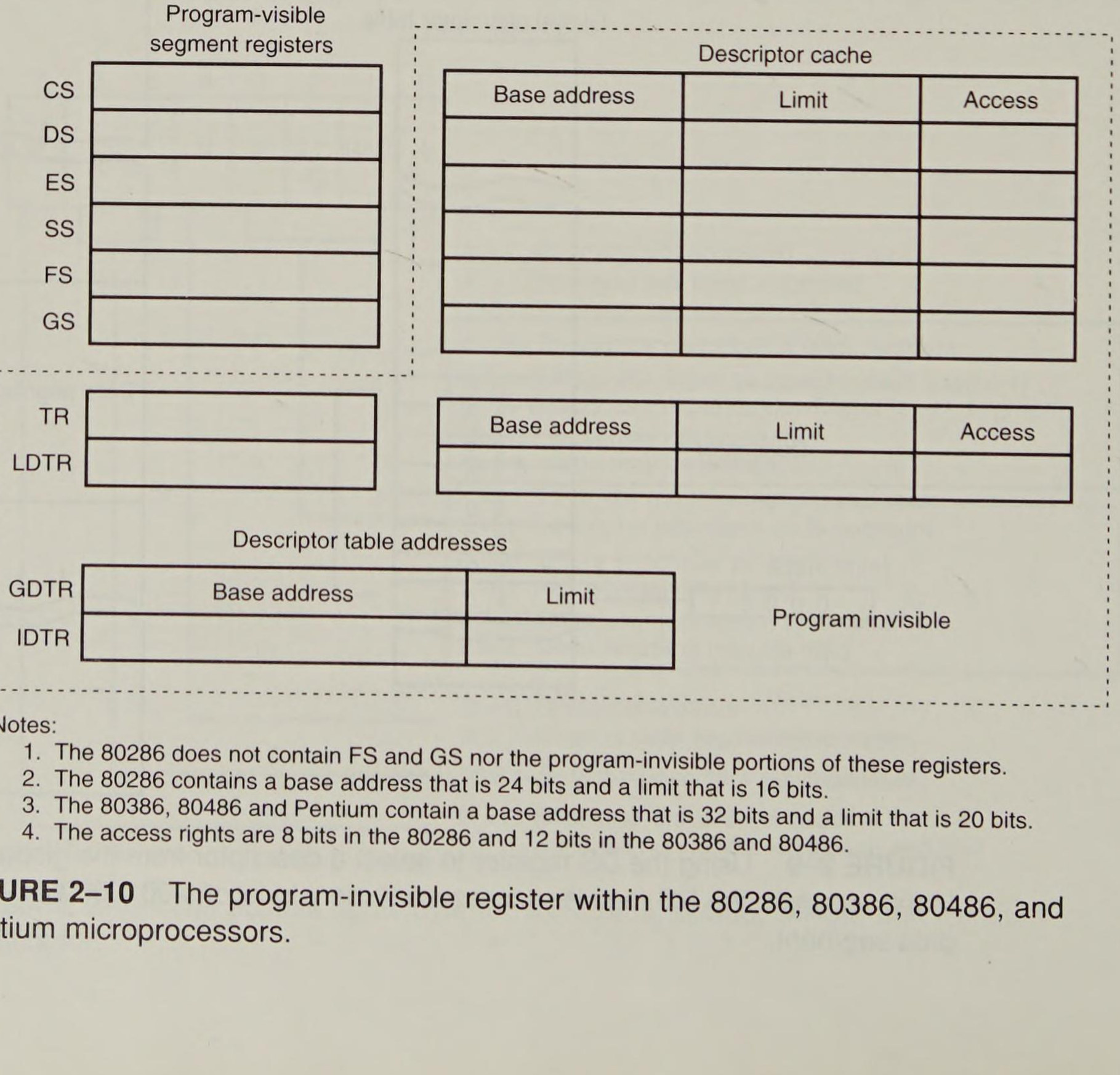

**FIGURE 2–9** Using the DS register to select a descriptor from the global descriptor table. In this example the DS register accesses memory locations 00100000H–001000FFH as a data segment.

in the memory system. In this illustration, DS contains 0008H, which accesses the descriptor number 1 from the global descriptor table using a requested privilege level of 00. Descriptor number 1 contains a descriptor that defines the base address as 00100000H with a segment limit of 000FFH. This means that a value of 0008H loaded into DS causes the microprocessor to use memory locations 00100000H–001000FFH for the data segment with this example descriptor table.

## Program-Invisible Registers

The global and local descriptor tables are found in the memory system. In order to access and specify the address of these tables, the 80286, 80386, 80486, and Pentium contain program-invisible registers. The *program-invisible registers* are not directly addressed by normal software so we give them this name. Some of these registers are accessed by the system software. Figure 2–10 illustrates the program-invisible registers as they appear in the 80286, 80386, 80486, and Pentium microprocessors. These registers control the microprocessor when operated in the protected mode.

Each of the segment registers contains a program-invisible portion used in the protected mode. This is a cache memory that is loaded with the base address, limit, and access rights each time the number is changed in the segment register. When a new segment number is placed in a segment register, the microprocessor accesses a descriptor table and loads the descriptor into the program-invisible cache portion of the segment register. It is




Notes:
1. The 80286 does not contain FS and GS nor the program-invisible portions of these registers.
2. The 80286 contains a base address that is 24 bits and a limit that is 16 bits.
3. The 80386, 80486 and Pentium contain a base address that is 32 bits and a limit that is 20 bits.
4. The access rights are 8 bits in the 80286 and 12 bits in the 80386 and 80486.

**FIGURE 2–10** The program-invisible register within the 80286, 80386, 80486, and Pentium microprocessors.

held there and used to access the memory segment until the segment number is again changed. This allows the microprocessor to repeatedly access a memory segment without referring back to the descriptor table for each access.

The GDTR (*global descriptor table register*) and IDTR (*interrupt descriptor table register*) contain the base address of the descriptor table and its limit. The limit of each descriptor table is 16 bits in the 80286, 80386, 80486, and Pentium microprocessors, because the maximum table length is 64K bytes. When protected-mode operation is desired, the address of the global descriptor table and its limit are loaded into the GDTR. Before using protected mode, the interrupt descriptor table and the IDTR must also be initialized. More detail is provided on protected-mode operation later in the text. At this point, the programming and additional description of these registers are impossible.

The location of the local descriptor table is selected from the global descriptor table. One of the global descriptors is set up to address the local descriptor table. To access the local descriptor table, the LDTR (*local descriptor table register*) is loaded with a selector just as a segment register is loaded with a selector. This selector accesses the global descriptor table and loads the base address, limit, and access rights of the local descriptor table into the cache portion of the LDTR.

The *task register* (TR) accesses a descriptor that defines a task. A task is most often a procedure or application program. The descriptor for the procedure or application program is stored in the global descriptor table so access can be controlled through the privilege levels. The task register allows a context or task switch in about 17 μs. Task switching allows the microprocessor to switch between tasks in a fairly short amount of time.

## 2–4 SUMMARY

1. The programming model of the 8086–80286 contains 8- and 16-bit registers. The programming model of the 80386, 80486, and Pentium contains 8-, 16-, and 32-bit extended registers as well as two additional 16-bit segment registers: FS and GS.
2. The 8-bit registers are: AH, AL, BH, BL, CH, CL, DH, and DL. The 16-bit registers are: AX, BX, CX, DX, SP, BP, DI, and SI. The segment registers are: CS, DS, ES, SS, FS, and GS. The 32-bit extended registers are: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. In addition, the microprocessor contains an instruction pointer (IP/EIP) and flag register (FLAGS or EFLAGS).
3. All real-mode memory addresses are a combination of a segment address plus an offset address. The starting location of a segment is defined by the 16-bit number in the segment register that is appended with a hexadecimal zero to its right-most end. The offset address is a 16-bit number added to the 20-bit segment address to form the real-mode memory address.
4. All instructions (code) are accessed by the combination of CS (segment address) plus IP or EIP (offset address).
5. Data are normally referenced through a combination of the DS (data segment) and either an offset address or the contents of a register that contains the offset address. The 8086–80486 and Pentium use BX, DI, and SI as default offset registers for data if

16-bit registers are selected. The 80386/80486/Pentium can use the 32-bit registers EAX, EBX, ECX, EDX, EDI, and ESI as default offset registers for data.

6. Protected-mode operation allows memory above the first 1M byte to be accessed by the 80286, 80386, 80486, and Pentium microprocessors. This extended memory system (XMS) is accessed via a segment address plus an offset address just as in the real mode. The difference is that the segment address is not held in the segment register. In the protected mode, the segment starting address is stored in a descriptor that is selected by the segment register.
7. A protected-mode descriptor contains a base address, limit, and access rights byte. The base address locates the starting address of the memory segment. The limit defines the last location of the segment. The access rights byte defines how the memory segment is accessed via a program. The 80286 microprocessor allows a memory segment to start at any of its 16M bytes of memory using a 24-bit base address. The 80386/80486/Pentium allows a memory segment to begin at any of its 4G bytes of memory using a 32-bit base address. The limit is a 16-bit number in the 80286 and a 20-bit number in the 80386/80486/Pentium. This allows an 80286 memory segment limit of 64K bytes and an 80386/80486/Pentium memory segment limit of either 1M bytes (G = 0) or 4G bytes (G = 1).
8. The segment register contains three fields of information in the protected mode. The left-most 13 bits of the segment register address one of 8,192 descriptors from a descriptor table. The TI bit accesses either the global descriptor table (TI = 0) or the local descriptor table (TI = 1). The right-most 2 bits of the segment register select the requested priority level for the memory segment access.
9. The program-invisible registers are used by the 80286–80486 and Pentium to access the descriptor tables. Each segment register contains a cache portion that is used in protected mode to hold the base address, limit, and access rights acquired from a descriptor. The cache allows the microprocessor to access the memory segment without again referring to the descriptor table until the segment register's contents are changed.

## 2–5 QUESTIONS AND PROBLEMS

1. What are program-visible registers?
2. The 80286 addresses registers that are 8- and _____-bits wide.
3. The extended registers are addressable by which microprocessors?
4. The extended BX register is addressed as _____.
5. Which register holds a count for some instructions?
6. What is the purpose of the IP/EIP register?
7. The carry flag bit is set by what arithmetic operations?
8. Will an overflow occur if a signed FFH is added to a signed 01H?
9. A number that contains three ones is said to have _____ parity.
10. Which flag bit controls the INTR pin on the microprocessor?
11. Which microprocessors contain an FS segment register?

12. What is the purpose of a segment register in the real-mode operation of the microprocessor?
13. In the real mode, show the starting and ending addresses of each segment located by the following segment register values:
    (a) 1000H
    (b) 1234H
    (c) 2300H
    (d) E000H
    (e) AB00H
14. Find the memory address of the next instruction executed by the microprocessor, when operated in the real mode, for the following CS:IP combinations:
    (a) CS = 1000H and IP = 2000H
    (b) CS = 2000H and IP = 1000H
    (c) CS = 2300H and IP = 1A00H
    (d) CS = 1A00H and IP = B000H
    (e) CS = 3456H and IP = ABCDH
15. Real-mode memory addresses allow access to memory below which address?
16. Which register or registers are used as an offset address for string instruction destinations in the 80286 microprocessor?
17. Which 32-bit register or registers are used as an offset address for data segment data in the 80386 microprocessor?
18. The stack memory is addressed by a combination of the _____ segment plus _____ offset.
19. If the base pointer (BP) address memory, the _____ segment contains the data.
20. Determine the memory location addressed by the following real-mode 80286 register combinations:
    (a) DS = 1000H and DI = 2000H
    (b) DS = 2000H and SI = 1002H
    (c) SS = 2300H and BP = 3200H
    (d) DS = A000H and BX = 1000H
    (e) SS = 2900H and SP = 3A00H
21. Determine the memory location addressed by the following real-mode 80386 register combinations:
    (a) DS = 2000H and EAX = 00003000H
    (b) DS = 1A00H and ECX = 00002000H
    (c) DS = C000H and ESI = 0000A000H
    (d) SS = 8000H and ESP = 00009000H
    (e) DS = 1239H and EDX = 0000A900H
22. Protected-mode memory addressing allows access to which area of the memory in the 80286 microprocessor?
23. Protected-mode memory addressing allows access to which area of the memory in the Pentium microprocessor?
24. What is the purpose of the segment register in protected-mode memory addressing?
25. How many descriptors are accessible in the global descriptor table in the protected mode?

26. For an 80286 descriptor that contains a base address of A00000H and a limit of 1000H, what starting and ending locations are addressed by this descriptor?
27. For an 80486 descriptor that contains a base address of 01000000H, a limit of 0FFFFH, and G = 0, what starting and ending locations are addressed by this descriptor?
28. For a Pentium descriptor that contains a base address of 00280000H, a limit of 00010H, and G = 1, what starting and ending locations are addressed by this descriptor?
29. If the DS register contains 0020H, in a protected-mode system, which global descriptor table entry is accessed?
30. If DS = 0103H, in a protected-mode system, the requested privilege level is _____.
31. If DS = 0105H, in a protected-mode system, which entry, table, and requested privilege level are selected?
32. What is the maximum length of the global descriptor table in the Pentium microprocessor?
33. Code a descriptor that describes a memory segment that begins at location 210000H and ends at location 21001FH. This memory segment is a code segment that can be read. The descriptor is for an 80286 microprocessor.
34. Code a descriptor that describes a memory segment that begins at location 03000000H and ends at location 05FFFFFFH. This memory segment is a data segment that grows upward in the memory system and can be written. The descriptor is for an 80386 microprocessor.
35. Which register locates the global descriptor table?
36. How is the local descriptor table addressed in the memory system?
37. Describe what happens when a new number is loaded into a segment register when the microprocessor is operated in the protected mode.
38. What are the program-invisible registers?
39. What is the purpose of the GDTR?

# CHAPTER 3

# Addressing Modes

## INTRODUCTION

Efficient software development for the 8086–80486 and Pentium requires a complete familiarity with the addressing modes employed by each instruction. In this chapter, we use the MOV (*move data*) instruction to describe the data-addressing modes. The MOV instruction transfers bytes or words of data between registers or between registers and memory in the 8086–80286 and bytes, words, or doublewords in the 80386, 80486, and Pentium microprocessors. In describing the program memory-addressing modes, we use the call and jump instructions that modify the flow of the program.

The data-addressing modes include: register, immediate, direct, register indirect, base-plus-index, register relative, and base relative-plus-index in the 8086–80286 microprocessor. The 80386, 80486, and Pentium also include a scaled-index mode of addressing memory data. The program memory-addressing modes include: program relative, direct, and indirect. The operation of the stack memory is explained so the PUSH and POP instructions are understood.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Explain the operation of each data-addressing mode.
2. Use the data-addressing modes to form assembly language statements.
3. Explain the operation of each program memory-addressing mode.
4. Use the program memory-addressing modes to form assembly and machine language statements.
5. Select the appropriate addressing mode to accomplish a given task.
6. Detail the difference between addressing memory data using real-mode and protected-mode operation.

7. Describe the sequence of events that place data onto the stack or remove data from the stack.
8. Explain how a data structure is placed in memory and used with software.

## 3–1 DATA-ADDRESSING MODES

Because the MOV instruction is a common and flexible instruction, it provides a basis for the explanation of the data-addressing modes. Figure 3–1 illustrates the MOV instruction and defines the direction of data flow. The source is to the right and the destination is to the left, next to the opcode MOV. (An *opcode* or *operation code* tells the microprocessor which operation to perform.) The direction of flow applied to all instructions is at first awkward, because we naturally assume things move from left to right, where here they move from right to left. Notice that a comma *always* separates the destination from source in an instruction.

In Figure 3–1, the MOV AX,BX instruction transfers the word contents of the source register (BX) into the destination register (AX). The source *never* changes, but the destination almost always changes.* It is useful to remember that a MOV instruction always *copies* the source data into the destination.

Figure 3–2 shows all variations of the data-addressing modes using the MOV instruction. This illustration helps to show how each data-addressing mode is formulated with the MOV instruction and also serves as a reference. Note that these are the same data-addressing modes found with all versions of the Intel microprocessor except for the scaled-index addressing mode which is only found in the 80386, 80486, and Pentium microprocessors. The data-addressing modes are:

1. *Register Addressing*—transfers a byte or word from the source register or memory location to the destination register or memory location. (Example: the MOV CX,DX instruction copies the word-sized contents of register DX into register CX.) In the 80386/80486/Pentium microprocessor, a doubleword can be transferred from the source register or memory location to the destination register or memory location. (Example: the MOV ECX,EDX instruction copies the doubleword-sized contents of register EDX into register ECX.)
2. *Immediate Addressing*—transfers an immediate byte or word of data into the destination register or memory location. (Example: the MOV AL,22H instruction copies a

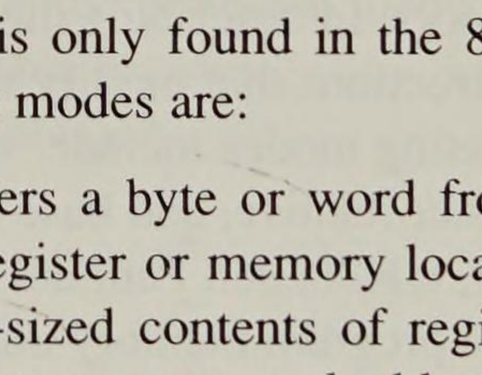



**FIGURE 3–1** The MOV instruction showing the source and destination and also direction of data flow.

*The exceptions are the CMP and TEST instructions that never change the destination. These instructions are described in later chapters.

| Type | Instruction | Source | Address Generation | Destination |
|---|---|---|---|---|
| Register | MOV AX,BX | Register BX | | Register AX |
| Immediate | MOV CH,3AH | Data 3AH | | Register CH |
| Direct | MOV [1234H],AX | Register AX | DS × 10H + DISP<br>10000H + 1234H | Memory address 11234H |
| Register indirect | MOV [BX],CL | Register CL | DS × 10H + BX<br>10000H + 0300H | Memory address 10300H |
| Base-plus-index | MOV [BX+SI],BP | Register SP | DS × 10H + BX + SI<br>10000H + 0300H + 0200H | Memory address 10500H |
| Register relative | MOV CL,[BX+4] | Memory address 10304H | DS × 10H + BX + 4<br>10000H + 0300H + 4 | Register CL |
| Base relative-plus-index | MOV ARRAY[BX+SI],DX | Register DX | DS × 10H + ARRAY + BX + SI<br>10000H + 1000H + 0300H + 0200H | Memory address 11500H |
| Scaled index | MOV [EBX+2 × ESI],AX | Register AX | DS × 10H + EBX + 2 × ESI<br>10000H + 00000300H + 00000400H | Memory address 10700H |

Notes: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

**FIGURE 3–2** The data-addressing modes.

byte-sized 22H into register AL.) In the 80386/80486/Pentium microprocessor, a doubleword of immediate data can be transferred into a register or memory location. (Example: the MOV EBX,12345678H instruction copies a doubleword-sized 12345678H into the 32-bit wide EBX register.)

3. *Direct Addressing*—moves a byte or word between a memory location and a register. (Example: the MOV CX,LIST instruction copies the word-sized contents of memory location LIST into register CX.) In the 80386/80486/Pentium a doubleword-sized memory location can also be addressed. (Example: the MOV ESI,LIST instruction copies a 32-bit number from 4 consecutive bytes of memory from location LIST into register ESI.)
4. *Register Indirect Addressing*—transfers a byte or word between a register and a memory location addressed by an index or base register. The index and base registers are: BP, BX, DI, and SI. (Example: the MOV AX,[BX] instruction copies the word-sized data from the data segment offset address indexed by BX into register AX.) In the 80386/80486/Pentium microprocessor a byte, word, or doubleword is transferred between a register and a memory location addressed by any register: EAX, EBX, ECX, EDX, EBP, EDI, or ESI. (Example: the MOV AL,[ECX] instruction loads AL from the data segment offset address selected by the contents of ECX.)
5. *Base-Plus-Index Addressing*—transfers a byte or word between a register and the memory location addressed by a base register (BP or BX) plus an index register (DI or SI). (Example: the MOV [BX+DI],CL instruction copies the byte-sized contents of register CL into the data segment memory location addressed by BX plus DI.) In the 80386/80486/Pentium any register EAX, EBX, ECX, EDX, EBP, EDI, or ESI may be combined to generate the memory address. (Example: the MOV [EAX+EBX],CL instruction copies the byte-sized contents of register CL into the data segment memory location addressed by EAX plus EBX.)
6. *Register Relative Addressing*—moves a byte or word between a register and the memory location addressed by an index or base register plus a displacement. (Example: MOV AX,[BX+4] or MOV AX,ARRAY[BX]. The first instruction loads AX from the data segment address formed by BX plus 4. The second instruction loads AX from the data segment memory location in ARRAY plus the contents of BX.) The 80386/80486/Pentium uses any register to address memory. (Example: MOV AX,[ECX+4] or MOV AX,ARRAY[EBX]. The first instruction loads AX, the data segment address formed by ECX plus 4. The second instruction loads AX from the data segment memory location ARRAY plus the contents of EBX.)
7. *Base Relative-Plus-Index Addressing*—transfers a byte or word between a register and the memory location addressed by a base and an index register plus a displacement. (Example: MOV AX,ARRAY[BX+DI] or MOV AX,[BX+DI+4]. These instructions both load AX from a data segment memory location. The first instruction uses an address formed by adding ARRAY, BX, and DI and the second by adding BX, DI, and 4.) (An 80386/80486/Pentium example: MOV EAX,ARRAY[EBX+ECX] loads EAX from the data segment memory location accessed by the sum of ARRAY, EBX, and ECX.)
8. *Scaled-Index Addressing*—is available only in the 80386, 80486, and Pentium microprocessors. The second register of a pair of registers is modified by the scale factor of 2X, 4X, or 8X to generate the operand memory address. (For example: a MOV EDX,[EAX+4*EBX] instruction loads EDX from the data segment memory location

addressed by EAX plus 4 *times* EBX.) Scaling allows access to word (2X), doubleword (4X), or quadword (8X) memory array data. Note that a scaling factor of 1X also exists, but it is normally implied and does not appear in the instruction. The MOV AL,[EBX+ECX] instruction is an example where the scaling factor is one. Alternately, we could rewrite the instruction as MOV AL,[EBX+1*ECX].

## Register Addressing

Register addressing is the most basic form of data addressing, once the register names are understood. The 8086–Pentium contains the following 8-bit registers used with register addressing: AH, AL, BH, BL, CH, CL, DH, and DL. They also contain the following 16-bit registers: AX, BX, CX, DX, SP, BP, SI, and DI. In the 80386/80486/Pentium, the extended 32-bit registers are: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. Some MOV instructions and the PUSH and POP instructions use the 16-bit segment registers (CS, ES, DS, SS, FS, and GS) for register addressing. It is important that instructions use registers that are of uniform size. *Never* mix an 8-bit register with a 16-bit register, an 8-bit register with a 32-bit register, or a 16-bit register with a 32-bit register because this is not allowed by the microprocessor. This is even true when a MOV AX,AL or a MOV EAX,AL instruction may seem to make sense. Of course the MOV AX,AL or MOV EAX,AL instruction is *not* allowed because these registers are of different sizes. Note that a few instructions take exception to this rule as indicated in later chapters.

Table 3–1 shows various versions of register MOV instructions. It is impossible to show all variations because there are so many possible combinations. For example, just the 8-bit subset of the MOV instruction has 64 different variations. About the only type of register MOV instruction not allowed is a segment-to-segment register MOV instruction. Also note that the code segment register may not be changed by a MOV instruction. The

**TABLE 3–1** Examples of the register-addressed instructions

| *Assembly Language* | *Operation* |
|---|---|
| MOV AL,BL | Copies BL into AL |
| MOV CH,CL | Copies CL into CH |
| MOV AX,CX | Copies CX into AX |
| MOV SP,BP | Copies BP into SP |
| MOV DS,AX | Copies AX into DS |
| MOV SI,DI | Copies DI into SI |
| MOV BX,ES | Copies ES into BX |
| MOV ECX,EBX | Copies EBX into ECX |
| MOV ESP,EDX | Copies EDX into ESP |
| MOV ES,DS | Not allowed (segment-to-segment) |
| MOV BL,BX | Not allowed (mixed sizes) |
| MOV CS,AX | Not allowed (the code segment register may not be used as a destination register) |

reason is that the address of the next instruction is found in both IP/EIP and CS. If only CS is changed, the address of the next instruction is unpredictable.

Figure 3–3 shows the operation of a MOV BX,CX instruction. Note that the source register's contents do not change, but the destination register's contents change. This instruction moves a 1234H from register CX into register BX. This data transfer erases the old contents (76AFH) of register BX, but the contents of CX remain unchanged. The contents of the destination register or destination memory location change for all instructions except the CMP and TEST instructions. Note that the MOV BX,CX instruction does not affect the left-most 16 bits of register EBX.

## Immediate Addressing

Another data-addressing mode is immediate addressing. The term *immediate* implies that the data immediately follow the hexadecimal opcode in the memory. Also note that immediate data is *constant data*, while the data transferred from a register is *variable data*. Immediate addressing operates on a byte or word of data. In the 80386/80486/Pentium microprocessor, immediate addressing also operates on doubleword data. The MOV immediate instruction transfers a copy of the immediate data into a register or a memory location. Figure 3–4 shows the operation of a MOV EAX,13456H instruction. This instruction copies the 13456H from the instruction, located in the memory immediately following the hexadecimal opcode, into register EAX. As with the MOV instruction illustrated in Figure 3–3, the source data overwrites the destination data.

In symbolic assembly language, the symbol # precedes immediate data in some assemblers.* The MOV AX,#3456H instruction is an example. Most assemblers do not use the # symbol but represent immediate data as in the MOV AX,3456H instruction. In this text we do not use the # symbol for immediate data. The most common assemblers—Intel ASM, Microsoft MASM,† and Borland TASM‡—do not use the # symbol

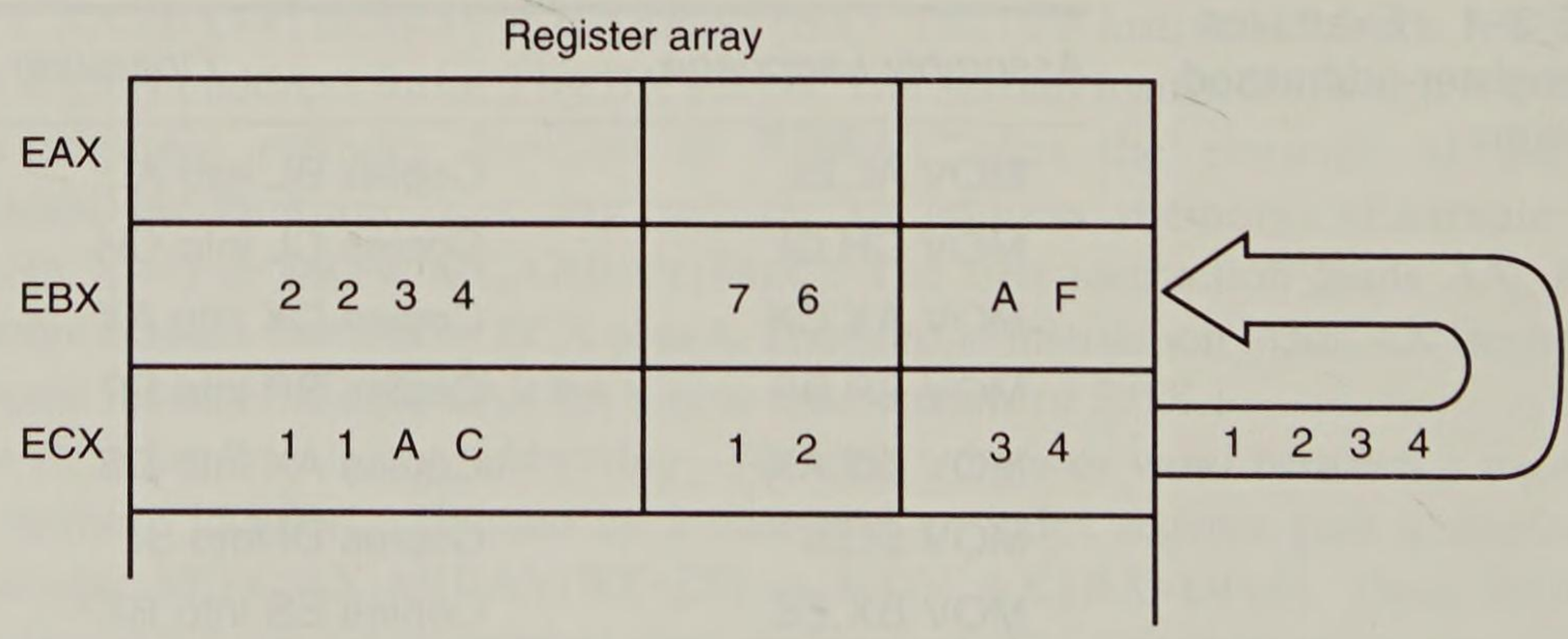



**FIGURE 3–3** The effect of executing the MOV BX,CX instruction at the point just before the BX register changes. Note that only the right-most 16 bits of register EBX change.

*This is true in the assembler provided for an HP64100 Logic Development system manufactured by Hewlett-Packard, Inc.

†The MASM (macro assembler) is an assembler program from Microsoft Corporation.

‡The TASM (turbo assembler) is an assembler program from Borland Corporation.

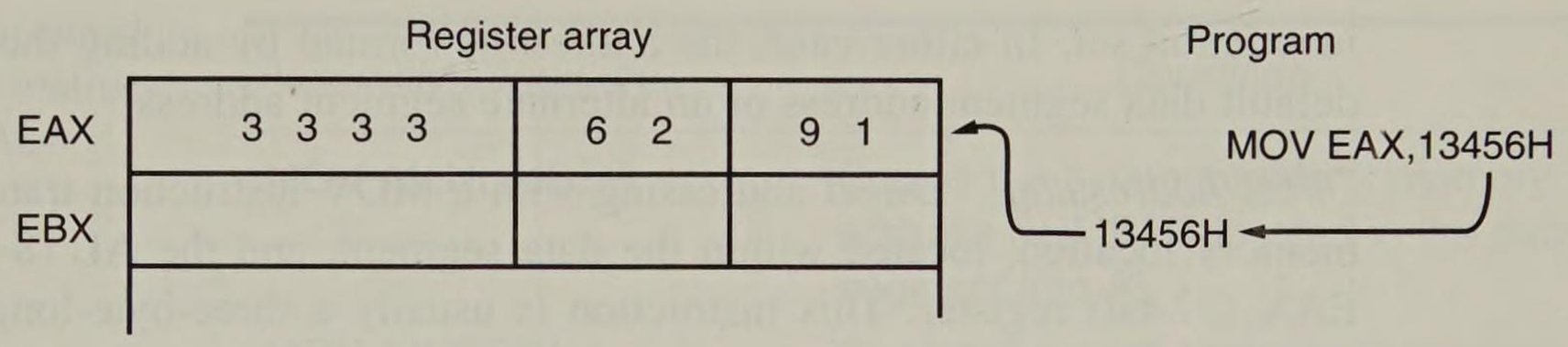



**FIGURE 3–4** The operation of the MOV EAX,13456H instruction. This instruction copies the immediate data (0013456H) into EAX.

for immediate data, but an older assembler used with the HP64000 logic development system does, as may others.

The symbolic assembler portrays immediate data in many ways. The letter H appends hexadecimal data. If hexadecimal data begins with a letter, we start it with a 0. For example, to represent a hexadecimal F2, we use 0F2H in assembly language. Decimal data is represented as is, and requires no special codes or adjustments. An example is the 100 decimal in the MOV AL,100 instruction. An ASCII-coded character or characters may be depicted in the immediate form if the ASCII data are enclosed in apostrophes. An example is the MOV BH,'A' instruction, which moves an ASCII-coded A (41H) into register BH. Be careful to use the apostrophe (') for ASCII data and not the single quotation mark (‘). Binary data are represented if the binary number is followed by the letter B, or, in some assemblers, the letter Y. Table 3–2 shows many different MOV instructions that apply immediate data.

## Direct Data Addressing

Most instructions can use the direct data addressing mode. Direct data addressing is applied to many instructions in a typical program. There are two basic forms of direct data addressing: (1) direct addressing that applies to a MOV between a memory location and AL, AX, or EAX, and (2) displacement addressing for almost any instruction in the

**TABLE 3–2** Examples of immediate addressing using the MOV instruction

| *Assembly Language* | *Operation* |
|---|---|
| MOV BL,44 | Moves a 44 decimal (2CH) into BL |
| MOV AX,44H | Moves a 44 hexadecimal into AX |
| MOV SI,0 | Moves a 0000H into SI |
| MOV CH,100 | Moves a 100 (64H) into CH |
| MOV AL,'A' | Moves an ASCII A (41H) into AL |
| MOV AX,'AB' | Moves an ASCII BA* (4241H) into AX |
| MOV CL,11001110B | Moves a binary 11001110 into CL |
| MOV EBX,12340000H | Moves a 12340000H into EBX |
| MOV ESI,12 | Moves a 12 decimal into ESI |
| MOV EAX,100Y | Moves a 100 binary into EAX |

**Note:* This is not an error, the ASCII characters are stored as a BA so care should be exercised when using a word-sized pair of ASCII characters.

instruction set. In either case, the address is formed by adding the displacement to the default data segment address or an alternate segment address.

*Direct Addressing.* Direct addressing with a MOV instruction transfers data between a memory location, located within the data segment, and the AL (8-bit), AX (16-bit), or EAX (32-bit) register. This instruction is usually a three-byte-long instruction. (In the 80386, 80486, and Pentium, a register size prefix may appear before the instruction causing it to exceed three bytes in length.)

The MOV AL,DATA instruction, as represented by most assemblers, loads AL from data segment memory location DATA (1234H). Memory location DATA is a *symbolic memory location*, while the 1234H is the actual hexadecimal location. With many assemblers we can also represent this instruction as a MOV AL,[1234H].* The [1234H] is an *absolute memory location* that is not allowed by all assembler programs. Figure 3–5 shows how this instruction transfers a copy of the byte-sized contents of memory location 11234H into AL. We form the effective address by adding 1234H (the offset address) to 10000H (the data segment address) in a system operating in the real mode.

Table 3–3 lists three direct-addressed instructions. These instructions often appear in programs so Intel decided to make them special three-byte-long instructions to reduce the length of programs. All other instructions that move data from a memory location to a register, called displacement addressed instructions, require four or more bytes of memory for storage in a program.

*Displacement Addressing.* Displacement addressing is almost identical with direct addressing except the instruction is four bytes wide instead of three. In the 80386/80486/Pentium, this instruction can be up to six bytes wide if a 32-bit register and displacement are specified. This type of direct data addressing is much more flexible because most instructions use it.

If the operation of the MOV CL,[1234H] instruction is compared to that of the MOV AL,[1234H] instruction of Figure 3–5, both basically perform the same operation except for the destination register (CL versus AL). Another difference only becomes apparent

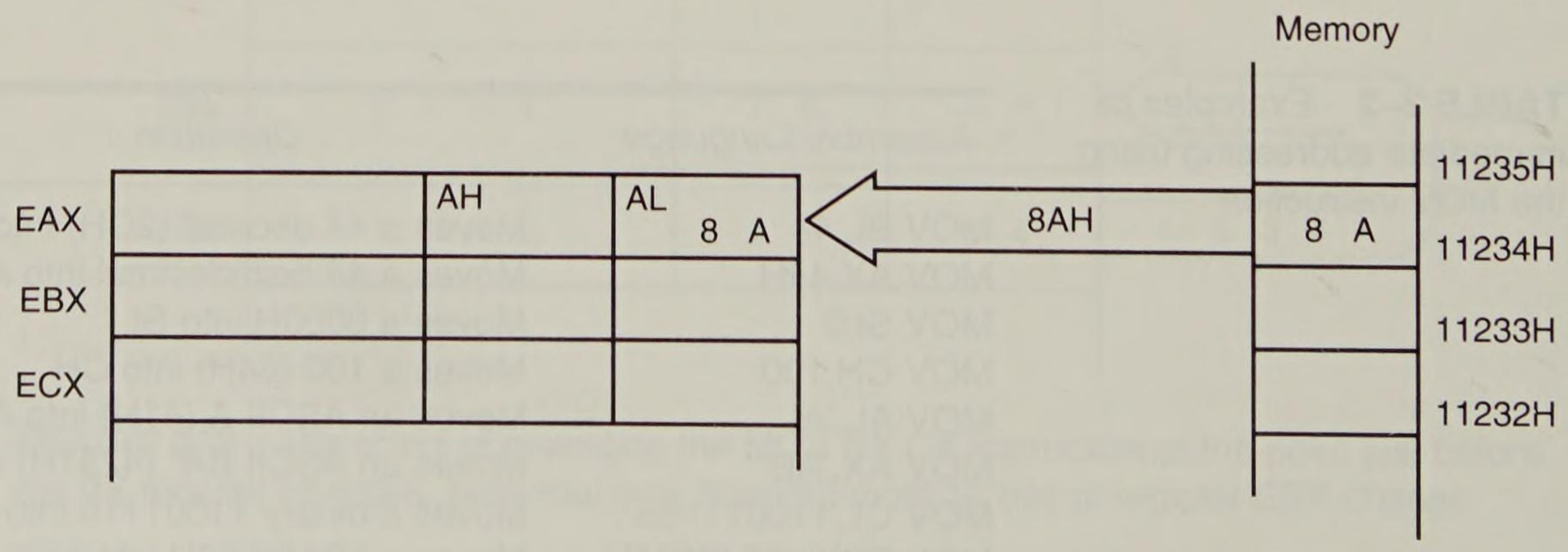



**FIGURE 3–5** The operation of the MOV AL,[1234H] instruction when DS = 1000H.

*This form may be used with MASM, but most often appears when a program is entered or listed by DEBUG, a debugging tool provided with the disk operating system.

**TABLE 3–3** The four possible direct-addressed instructions using AX and AL

| *Assembly Language* | *Operation* |
|---|---|
| MOV AL,NUMBER | Copies the byte contents of memory address NUMBER located in the data segment into AL |
| MOV AX,COW | Copies the word contents of memory address COW in the data segment into AX |
| *MOV EAX,WATER | Copies the doubleword contents of memory WATER located in the data segment into EAX |
| MOV NEWS,AL | Copies AL into memory location NEWS in the data segment |
| MOV THERE,AX | Copies AX into memory location THERE in the data segment |
| *MOV HOME,EAX | Copies EAX into data segment memory location HOME |

**Note*: The 80386/80486/Pentium microprocessor will at times use more than three bytes of memory for the 32-bit move between EAX and memory.

upon examination of the assembled versions of these two instructions. The MOV AL,[1234H] instruction is three bytes in length, and the MOV CL,[1234H] instruction is four bytes, as illustrated in Example 3–1. This example shows how the assembler converts these two instructions into hexadecimal machine language.

**EXAMPLE 3–1**

```
0000 A0 1234 R                  MOV  AL,[1234H]
0003 8A 0E 1234 R               MOV  CL,[1234H]
```

Table 3–4 lists some MOV instructions using the displacement form of direct addressing. Not all variations are listed because there are many MOV instructions of this type. Please notice that the segment registers can be stored or loaded from memory.

## Register Indirect Addressing

Register indirect addressing allows data to be addressed at any memory location through the offset address held in any of the following registers: BP, BX, DI, and SI. For example, if register BX contains a 1000H and the MOV AX,[BX] instruction executes, the word contents of data segment memory offset address 1000H are copied into register AX. If the microprocessor is operated in the real mode and DS = 0100H, this instruction addresses a word stored at memory bytes 2000H and 2001H and transfers them into register AX (see Figure 3–6). Note that the contents of 2000H are moved into AL and the contents of 2001H are moved into AH. The [ ] symbols denote indirect addressing in assembly language. In addition to using the BP, BX, DI, and SI registers to indirectly address memory,

**TABLE 3–4** Examples of direct data addressing using a displacement

| *Assembly Language* | *Operation* |
|---|---|
| MOV CH,DOG | Loads CH with the contents of data segment memory location DOG (the actual offset address of DOG is calculated by the assembler) |
| MOV CH,[1000H] | *Loads CH with the contents of data segment memory location 1000H |
| MOV ES,DATA6 | Loads ES with the word-sized contents of data segment memory location DATA6 |
| MOV DATAS,BP | BP is copied into data segment memory location DATAS |
| MOV NUMBER,SP | SP is copied into data segment memory location NUMBER |
| MOV DATA1,EAX | EAX is copied into data segment memory location DATA1 |
| MOV EDI,SUM1 | Loads EDI with the contents of data segment memory location SUM1 |

**Note*: This form of addressing is seldom used with most assemblers because we don't normally address an actual numeric offset address.

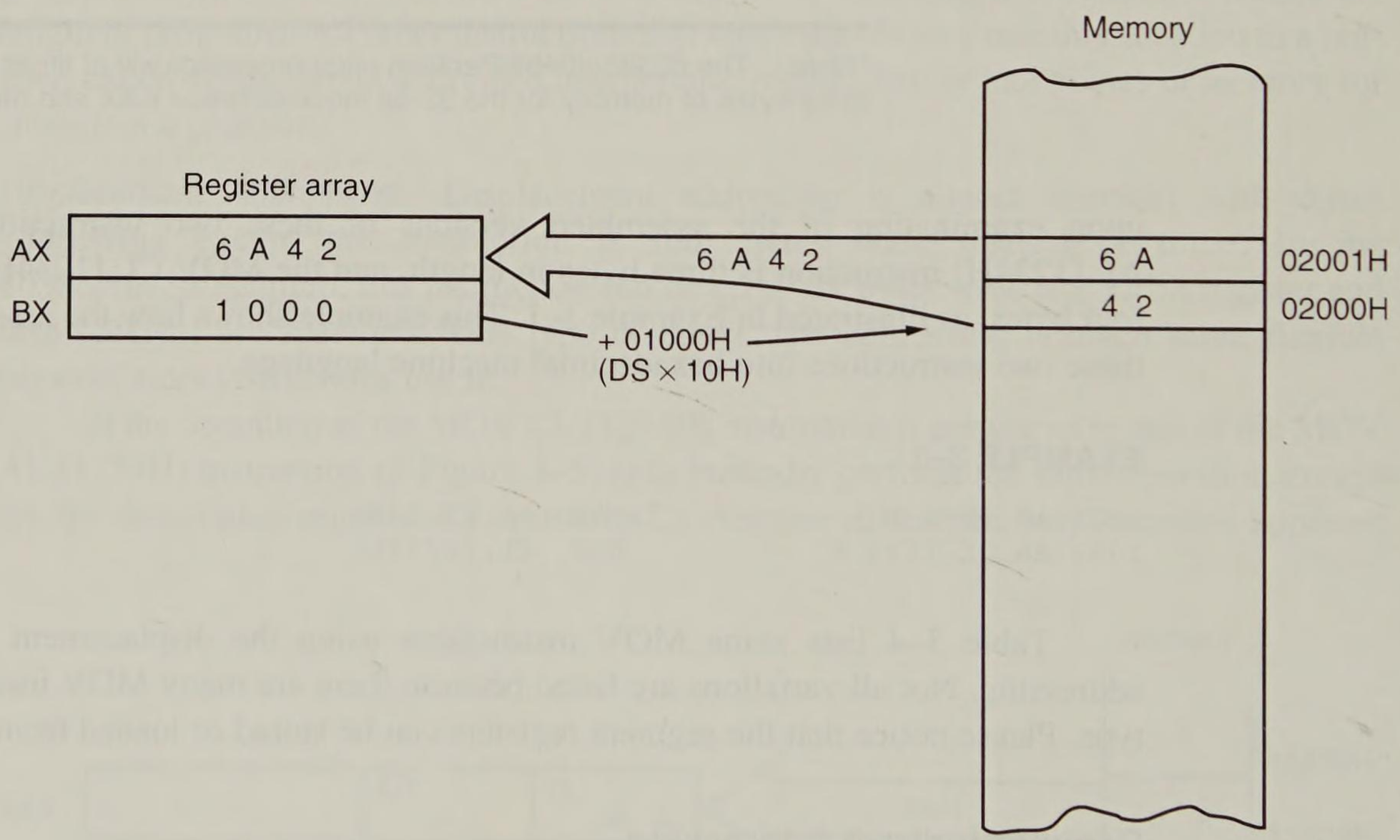



**FIGURE 3–6** The effect of executing the MOV AX,[BX] instruction if DS = 0100H and BX = 1000H.

the 80386/80486/Pentium allows register indirect addressing with any extended register except ESP. Some typical instructions using indirect addressing appear in Table 3–5.

The data segment is used by default with register indirect addressing or any other addressing mode that uses BX, DI, or SI to address memory. If register BP addresses memory, the stack segment is used by default. These are considered the default settings for these four index and base registers. For the 80386, 80486, and Pentium, EBP addresses

**TABLE 3–5** Example instructions using register indirect addressing

| *Assembly Language* | **Operation* |
|---|---|
| MOV CX,[BX] | CX is loaded with a word from the data segment memory location addressed by BX |
| MOV [BP],DL | A byte is copied from register DL into the stack segment memory location addressed by BP |
| MOV [DI],BH | A byte is copied from register BH into the data segment memory location addressed by DI |
| MOV [DI],[BX] | Memory-to-memory moves are not allowed except with string instructions |
| MOV DI,[DI] | This instruction is not allowed because the register used to indirectly address memory may not be changed by the instruction |
| MOV AL,[EDX] | AL is loaded from the data segment memory location addressed by EDX |
| MOV ECX,[EBX] | ECX is loaded with a doubleword from the data segment memory location addressed by EBX |

**Note*: Data addressed by BP or EBP is by default located in the stack segment, while all other indirect addressing modes use the data segment by default.

memory, by default, in the stack segment while EAX, EBX, ECX, EDX, EDI, and ESI address memory in the data segment by default. When using a 32-bit register to address memory in the real mode, the contents of the 32-bit register must never exceed 0000FFFFH. In the protected mode, any value can be used in a 32-bit register used to indirectly address memory as long as it does not access a location outside of the segment as dictated by the access rights byte. An example 80386/80486/Pentium instruction is MOV EAX,[EBX]. This instruction loads EAX with the doubleword-sized number stored at the data segment offset address indexed by EBX.

In some cases, indirect addressing requires that the size of the data is specified with the special assembler directive BYTE PTR, WORD PTR, or DWORD PTR. These directives indicate the size of the memory data addressed by the *memory pointer* (PTR). For example, the MOV AL,[DI] instruction is clearly a byte-sized MOV instruction, but the MOV [DI],10H instruction is ambiguous. Does the MOV [DI],10H instruction address a byte-, word-, or doubleword-sized memory location? We can't determine the size and neither can the assembler. The instruction MOV BYTE PTR [DI],10H clearly designates the location addressed by DI as a byte-sized memory location. Likewise, the MOV DWORD PTR [DI],10H clearly identifies the memory location as doubleword-sized. The BYTE PTR, WORD PTR, and DWORD PTR directives are only used with instructions that address a memory location through a pointer or index register with immediate data and for a few other instructions that are described in subsequent chapters.

Indirect addressing often allows a program to refer to tabular data located in the memory system. For example, suppose that you must create a table of information that contains 50 samples taken from a digital voltmeter. Figure 3–7 shows the table and the BX register used to address each location in the table sequentially. To do this, you must load the starting location of the table into the BX register with a MOV immediate instruction.

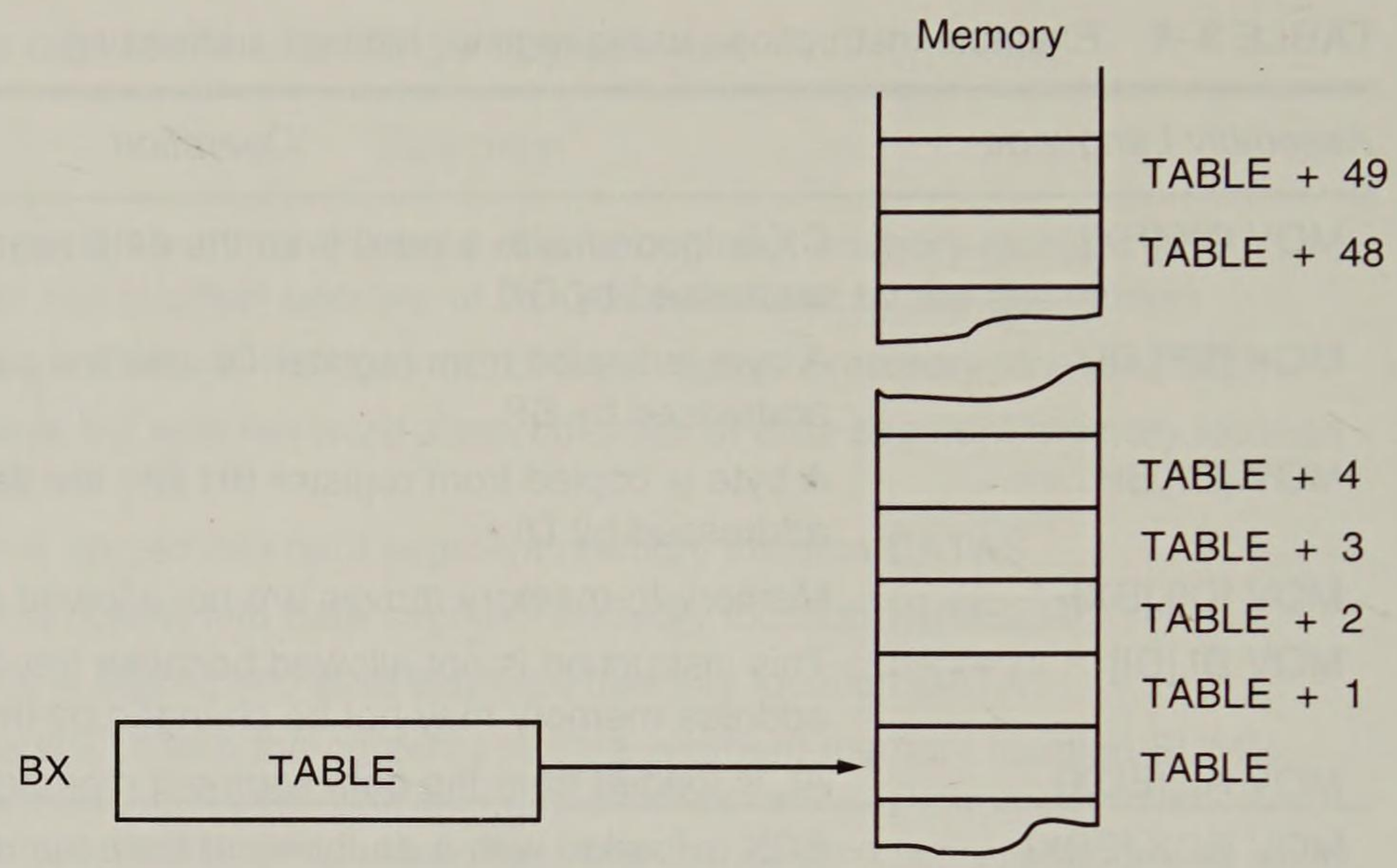



**FIGURE 3–7** An array of data (TABLE) containing 50 bytes that are indirectly addressed through the BX register.

After initializing the starting address of the table, use register indirect addressing to store the 50 samples sequentially.

**EXAMPLE 3–2**

```
                        ;Instructions that read 50 bytes of data from the DATA_PORT
                        ;and stores them in array TABLE
                        ;
0000 BB 0000 R                MOV  BX,OFFSET TABLE ;address TABLE
0003 B9 0032                  MOV  CX,50           ;load counter
0006                    AGAIN:
0006 E4 2A                    IN   AL,DATA_PORT    ;read voltmeter
0008 88 07                    MOV  [BX],AL         ;save data
000A 43                       INC  BX              ;address next
000B E2 F9                    LOOP AGAIN           ;repeat 50 times
```

The sequence shown in Example 3–2 loads register BX with the starting address of the table and initializes the count, located in register CX, to 50. The OFFSET directive tells the assembler to load BX with the offset address of memory location TABLE and not the contents of TABLE. For example, the MOV BX,DATA instruction copies the contents of memory location DATA into BX while the MOV BX,OFFSET DATA instruction copies the address of DATA into BX. When the OFFSET directive is used with the MOV instruction, the assembler calculates the offset address and then uses a MOV immediate instruction to load the address into the specified 16-bit register.

Once the counter and pointer are initialized, a repeat-until CX = 0 loop executes. Here data are input (IN) from the voltmeter and stored in the memory location indirectly addressed by the offset address located in register BX. Next, BX increments (*adds one*) to the next table location and finally the LOOP instruction repeats the LOOP 50 times. The LOOP instruction decrements (*subtracts one*) the counter (CX) and if CX is not zero, LOOP causes a jump to memory location AGAIN. If CX becomes zero, no jump occurs and this sequence of instructions ends. This example makes no attempt to show synchronization between the voltmeter and the microprocessor.

## Base-Plus-Index Addressing

Base-plus-index addressing is similar to indirect addressing because it indirectly addresses memory data. In the 8086–80286, this type of addressing uses one base register (BP or BX) and one index register (DI or SI) to indirectly address memory. Often the base register holds the beginning location of a memory array, while the index register holds the relative position of an element in the array. Remember that whenever BP addresses memory data, both the stack segment register and BP generate the effective address.

In the 80386/80486/Pentium, this type of addressing allows the combination of any two 32-bit extended registers except ESP. For example, the MOV DL,[EAX+EBX] instruction is an example using EAX (as the base) plus EBX (as the index). If the EBP register is used, the data are located in the stack segment instead of the data segment.

*Locating Data with Base-Plus-Index addressing.* Figure 3–8 shows how data are addressed by the MOV DX,[BX+DI] instruction when the microprocessor operates in the real mode. In this example, BX = 1000H, DI = 0010H, and DS = 0100H, which translate into memory address 02010H. This instruction transfers a copy of the word from location 02010H into the DX register. Table 3–6 lists some instructions used for base-plus-index addressing. Note that the Intel assembler requires that this addressing mode appears as [BX][DI] instead of [BX+DI]. The MOV DX,[BX+DI] instruction is MOV DX,[BX][DI] for a program written for the Intel ASM assembler. This text uses the first form in all example programs, but the second form can be used in many assemblers including MASM from Microsoft.

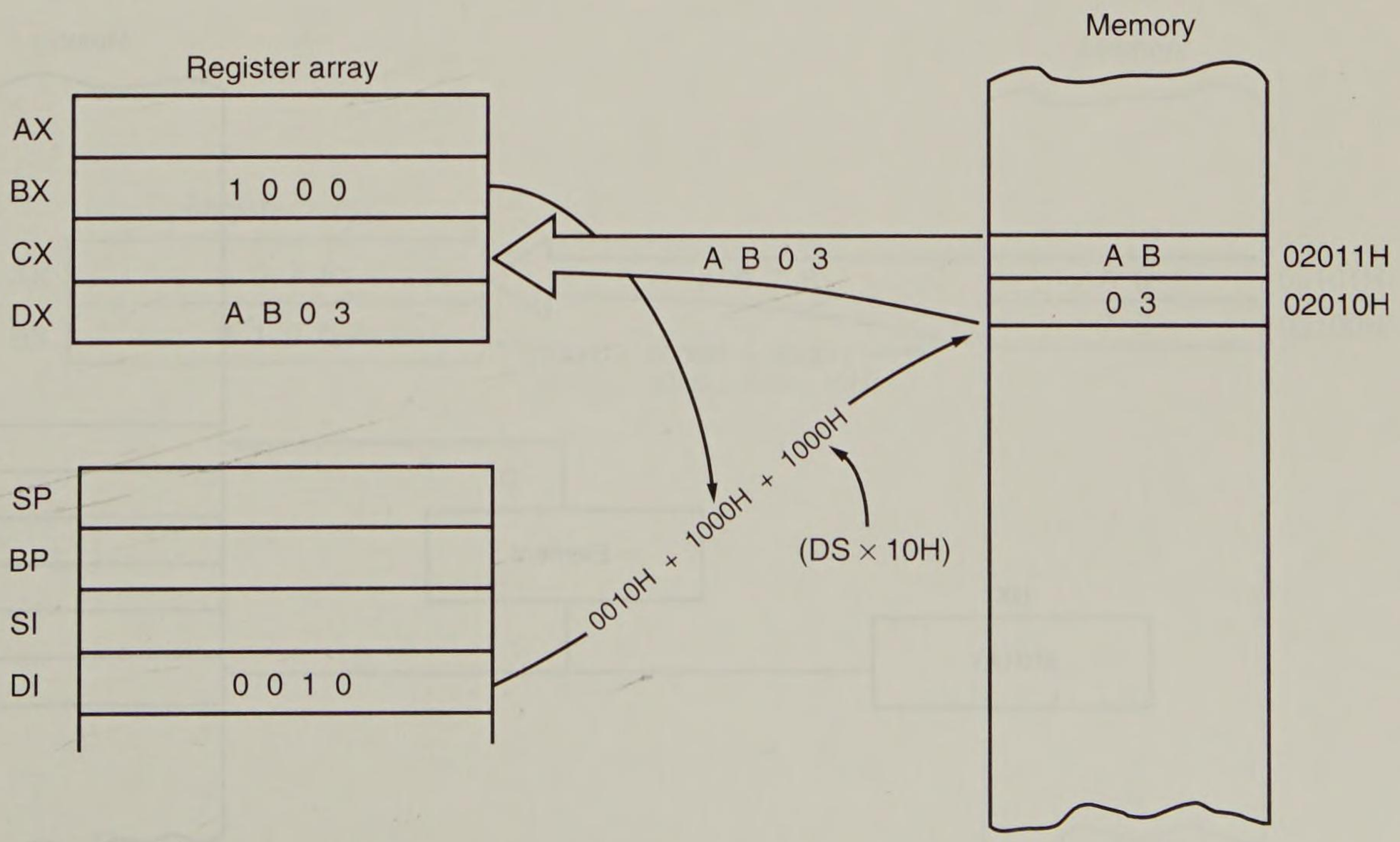



**FIGURE 3–8** An example showing how the base-plus-index mode of addressing functions for the MOV DX,[BX + DI] instruction. Notice that memory address 02010H is referenced by this instruction because DS (0100H), BX (1000H), and DI (0010H) are summed to generate this address.

**TABLE 3–6** Examples of base-plus-index addressing

| *Assembly Language* | *Operation* |
|---|---|
| MOV CX,[BX+DI] | Loads CX with the word contents of the data segment memory location addressed by BX plus DI |
| MOV CH,[BP+SI] | Loads CH with the byte contents of the stack segment memory location addressed by BP plus SI |
| MOV [BX+SI],SP | Stores the word contents of SP into the data segment memory location addressed by BX plus SI |
| MOV [BP+DI],CX | Stores the word contents of CX into the stack segment memory location addressed by BP plus DI |
| MOV CL,[EDX+EDI] | Loads CL with the byte contents of the data segment memory location addressed by EDX plus EDI |
| MOV [EAX+EBX],ECX | Stores the doubleword contents of ECX into the data segment memory location addressed by EAX plus EBX |

*Locating Array Data Using Base-Plus-Index Addressing.* A major use of the base-plus-index addressing mode is to address elements in a memory array. Suppose we must address the elements in an array located in the data segment at memory location ARRAY. To accomplish this, we load the BX register (base) with the beginning address of the array and the DI register (index) with the element number to be accessed. Figure 3–9 shows the use of BX and DI to access an element in an array of data.

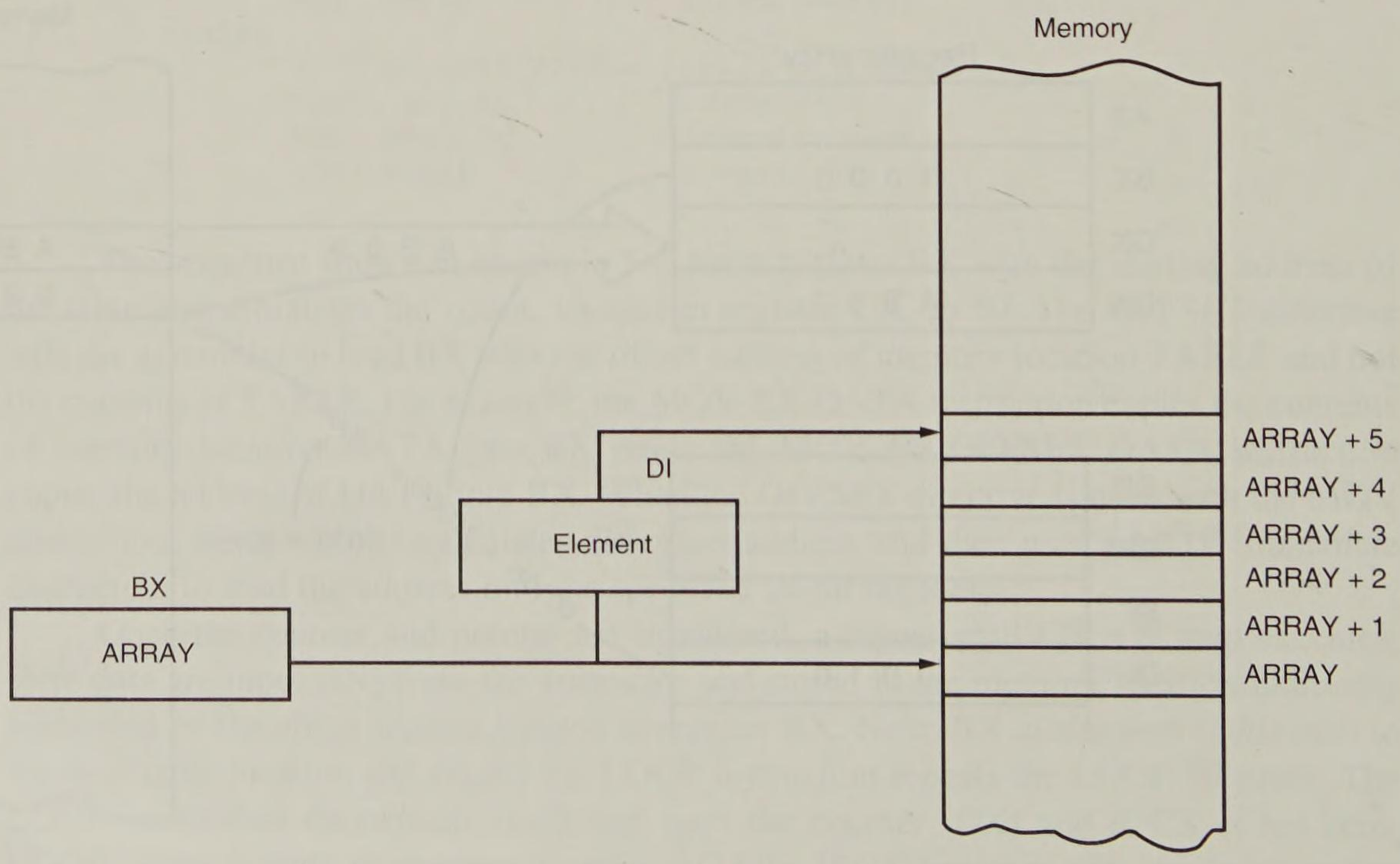



**FIGURE 3–9** An example showing base-plus-index addressing. Here an element (DI) of an array of data (BX) is accessed.

**EXAMPLE 3–3**

```
                        ;Using the base-plus-index addressing mode
                        ;
0000 BB 0000 R                MOV  BX,OFFSET ARRAY ;address ARRAY
0003 BF 0010                  MOV  DI,10H          ;element 10H
0006 8A 01                    MOV  AL,[BX+DI]      ;get data
0008 BF 0020                  MOV  DI,20H          ;element 20H
000B 88 01                    MOV  [BX+DI],AL      ;save data
```

A short program listed in Example 3–3 moves array element 10H into array element 20H. Notice that the array element number, loaded into the DI register, addresses the array element.

## Register Relative Addressing

Register relative addressing is similar to base-plus-index addressing and displacement addressing discussed earlier. In register relative addressing, the data in a segment of memory are addressed by adding the displacement to the contents of a base or an index register (BP, BX, DI, or SI). Figure 3–10 shows the operation of the MOV AX,[BX+1000H] instruction. In this example, BX = 0100H and DS = 0200H so the address generated is the sum of DS × 10H, BX, and the displacement of 1000H or 03100H. Remember that BX, DI, or SI address the data segment and BP addresses the stack segment. In the 80386/80486 the displacement can be a 32-bit number and the register can be any 32-bit register except the ESP register. Remember that the size of a real-mode segment is 64K bytes in length. Table 3–7 lists a few instructions that use register relative addressing.

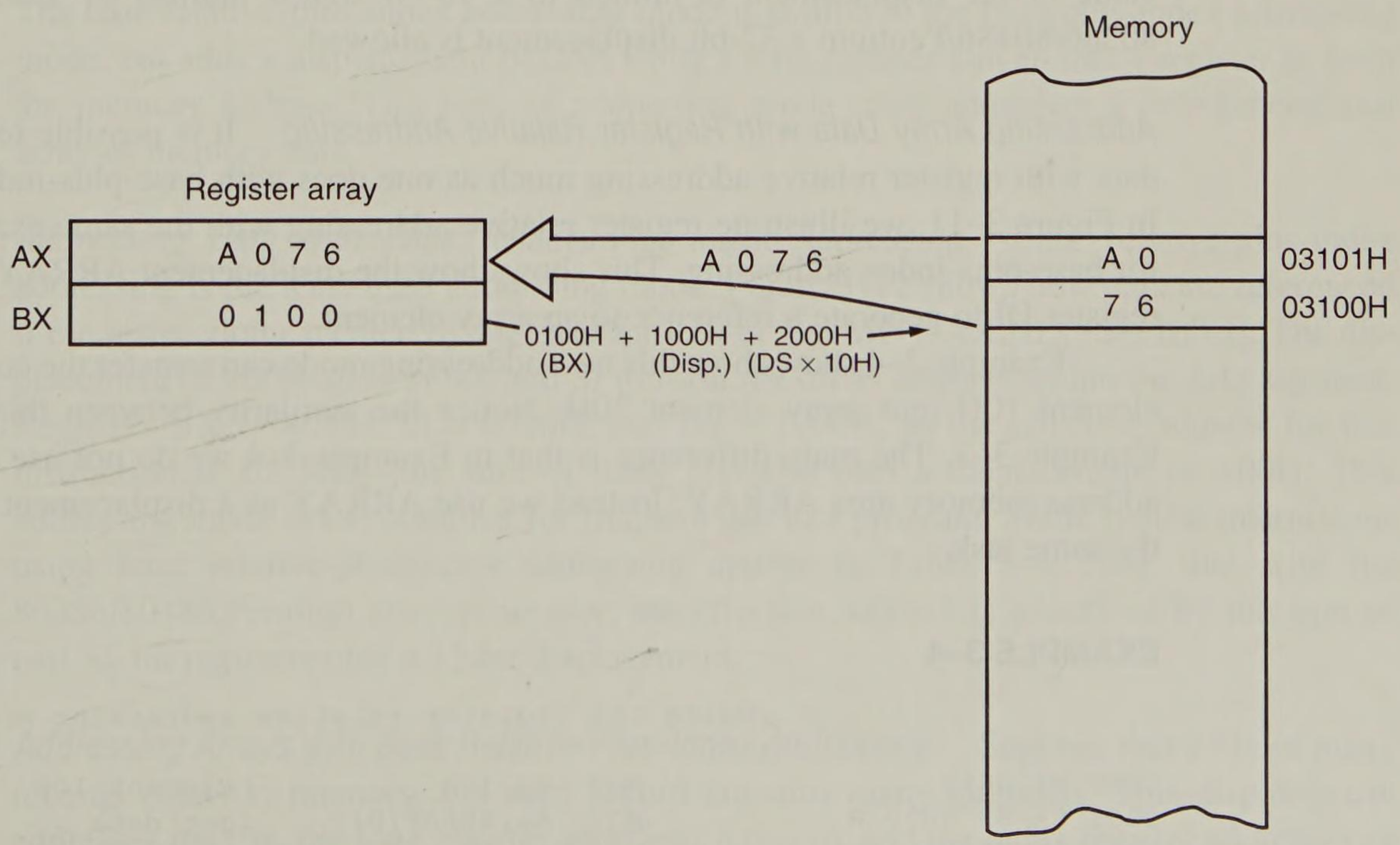



**FIGURE 3–10** The effect of executing the MOV AX,[BX + 1000H] instruction if BX = 0100H and DS = 0200H.

**TABLE 3–7** Examples of register relative addressing

| *Assembly Language* | *Operation* |
|---|---|
| MOV AX,[DI+100H] | Loads AX from the word contents of the data segment memory location addressed by DI plus 100H |
| MOV ARRAY[SI],BL | Stores the byte contents of BL into the data segment memory location addressed by ARRAY plus SI |
| MOV LIST[SI+2],CL | Stores the byte contents of CL into the data segment memory location addressed by the sum of LIST, SI, and 2 |
| MOV DI,SETS[BX] | Loads DI from the data segment memory location addressed by SETS plus BX |
| MOV DI,[EAX+100H] | Loads DI from the data segment memory location addressed by EAX plus 100H |
| MOV ARRAY[EBX],AL | Stores AL into the data segment memory location address by ARRAY plus EBX |

The displacement can be a number added to the register within the [ ], as in the MOV AL,[DI+2] instruction, or it can be a displacement subtracted from the register, as in MOV AL,[SI–1]. A displacement also can be an offset address appended to the front of the [ ] as in MOV AL,DATA[DI]. Both forms of displacements also can appear simultaneously as in the MOV AL,DATA[DI+3] instruction. In all cases both forms of the displacement add to the base or base and index register within the [ ]. In the 8086–80286 microprocessor the value of the displacement is limited to a 16-bit signed number of ± 32K and in the 80386/80486/Pentium a 32-bit displacement is allowed.

*Addressing Array Data with Register Relative Addressing.* It is possible to address array data with register relative addressing much as one does with base-plus-index addressing. In Figure 3–11, we illustrate register relative addressing with the same example we used for base-plus-index addressing. This shows how the displacement ARRAY adds to index register DI to generate a reference to an array element.

Example 3–4 shows how this new addressing mode can transfer the contents of array element 10H into array element 20H. Notice the similarity between this example and Example 3–3. The main difference is that in Example 3–4 we do not use register BX to address memory area ARRAY; instead we use ARRAY as a displacement to accomplish the same task.

**EXAMPLE 3–4**

```
                        ;Using the register relative addressing mode
                        ;
0000 BF 0010                    MOV  DI,10H           ;element 10H
0003 8A 85 0000 R               MOV  AL,ARRAY[DI]     ;get data
0007 BF 0020                    MOV  DI,20H           ;element 20H
000A 88 85 0000 R               MOV  ARRAY[DI],AL     ;save data
```

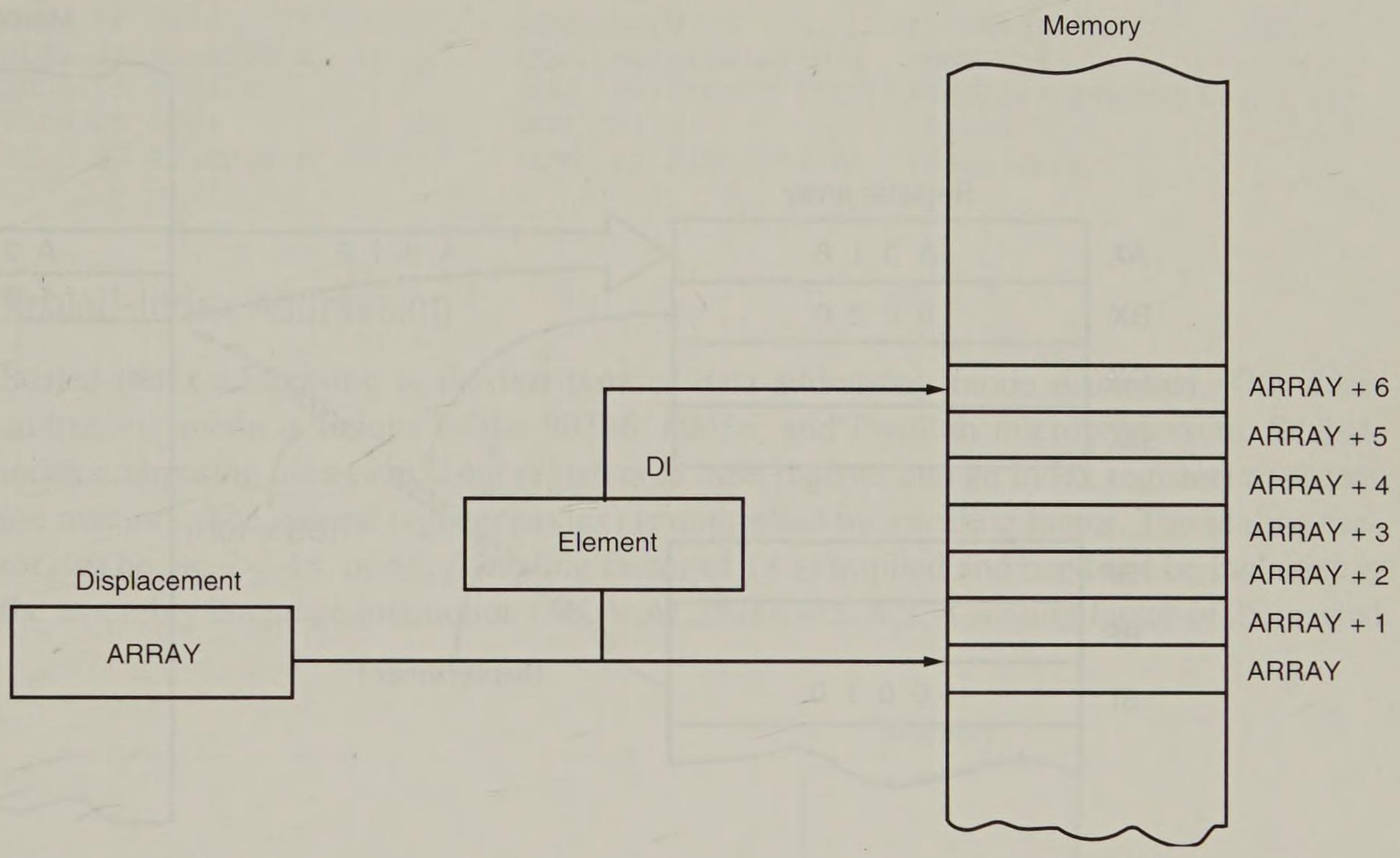



**FIGURE 3–11** Register relative addressing used to address an element of ARRAY. The displacement addresses the start of the array, and the contents of DI (the element) selects an element of the array.

## Base Relative-Plus-Index Addressing

The base relative-plus-index addressing mode is similar to the base-plus-index addressing mode, but adds a displacement besides using a base register and an index register to form the memory address. This type of addressing mode often addresses a two-dimensional array of memory data.

*Addressing Data with Base Relative-Plus-Index Addressing.* Base relative-plus-index addressing is the least-used addressing mode. Figure 3–12 shows how data are referenced if the instruction executed by the microprocessor is a MOV AX,[BX+SI+100H]. The displacement of 100H adds to BX and SI to form the offset address within the data segment. Registers BX = 0020H, SI = 0010H, and DS = 1000H, so the effective address for this instruction is 10130H—the sum of these registers plus a displacement of 100H. This addressing mode is too complex for frequent use in a program. Some typical instructions using base relative-plus-index addressing appear in Table 3–8. Note that with the 80386/80486/Pentium microprocessor, the effective address is generated by the sum of two 32-bit registers plus a 32-bit displacement.

*Addressing Arrays with Base Relative-Plus-Index Addressing.* Suppose that a file of many records exists in memory and each record contains many elements. This displacement addresses the file, the base register addresses a record, and the index register addresses an element of a record. Figure 3–13 illustrates this very complex form of addressing.

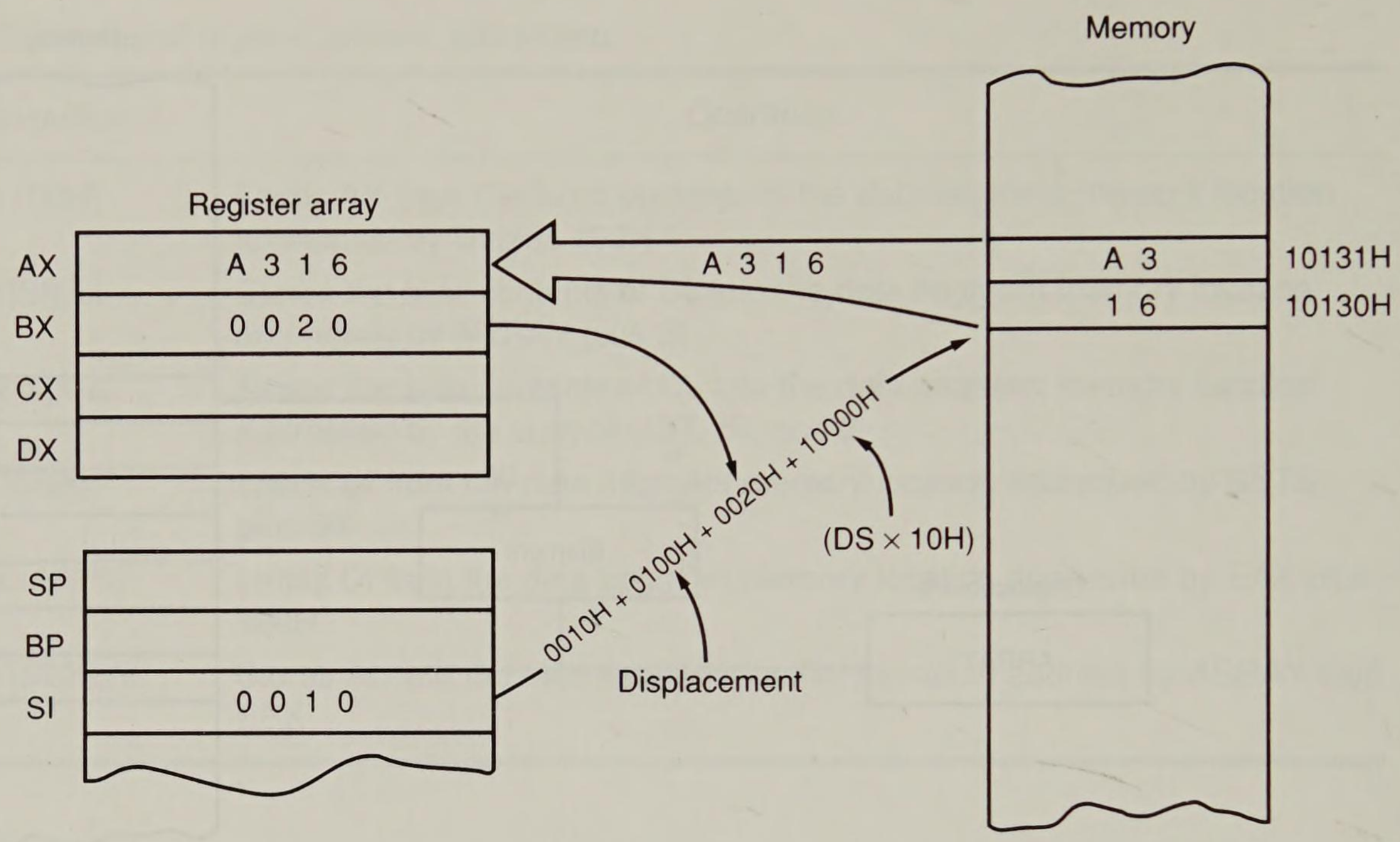



**FIGURE 3–12** An example of base relative-plus-index addressing is illustrated with the MOV 100H[BX + SI] instruction. This instruction moves data from memory to the AX register. The memory address is the sum of DS × 10H, 100H, BX, and SI.

**TABLE 3–8** Example base relative-plus-index instructions

| *Assembly Language* | *Operation* |
|---|---|
| MOV DH,[BX+DI+20H] | Loads DH from the data segment memory location addressed by the sum of BX, DI, and 20H |
| MOV AX,FILE[BX+DI] | Loads AX from the data segment memory location addressed by the sum of FILE, BX, and DI |
| MOV LIST[BP+DI],CL | Stores CL at the stack segment memory location addressed by the sum of LIST, BP, and DI |
| MOV LIST[BP+SI+4],DH | Stores DH at the stack segment memory location addressed by the sum of LIST, BP, SI, and 4 |
| MOV AL,FILE[EBX+ECX+2] | Loads AL from the data segment memory location addressed by the sum of FILE, EBX, ECX, and 2 |

Example 3–5 provides a program that copies element 0 of record A into element 2 of record C using the base relative-plus-index mode of addressing.

**EXAMPLE 3–5**

```
                        ;Using the base relative-plus-index addressing mode
                        ;
0000 BB 0000 R                  MOV  BX,OFFSET RECA  ;address record A
```

```
0003 BF 0000                MOV  DI,0              ;element 0
0006 8A 81 0000 R           MOV  AL,FILE[BX+DI]    ;get data
000A BB 0064 R              MOV  BX,OFFSET RECC    ;address record C
000D BF 0002                MOV  DI,2              ;element 2
0010 88 81 0000 R           MOV  FILE[BX+DI],AL    ;save data
```

## Scaled-Index Addressing

Scaled-index addressing is the last type of data-addressing mode discussed. This data-addressing mode is unique to the 80386, 80486, and Pentium microprocessors. Scaled-index addressing uses two 32-bit registers (a base register and an index register) to access the memory. The second register (index) is multiplied by a scaling factor. The scaling factor can be 1×, 2×, 4×, or 8×. A scaling factor of 1× is implied and need not be included in the assembly language instruction (MOV AL,[EBX+ECX]). A scaling factor of 2× is used

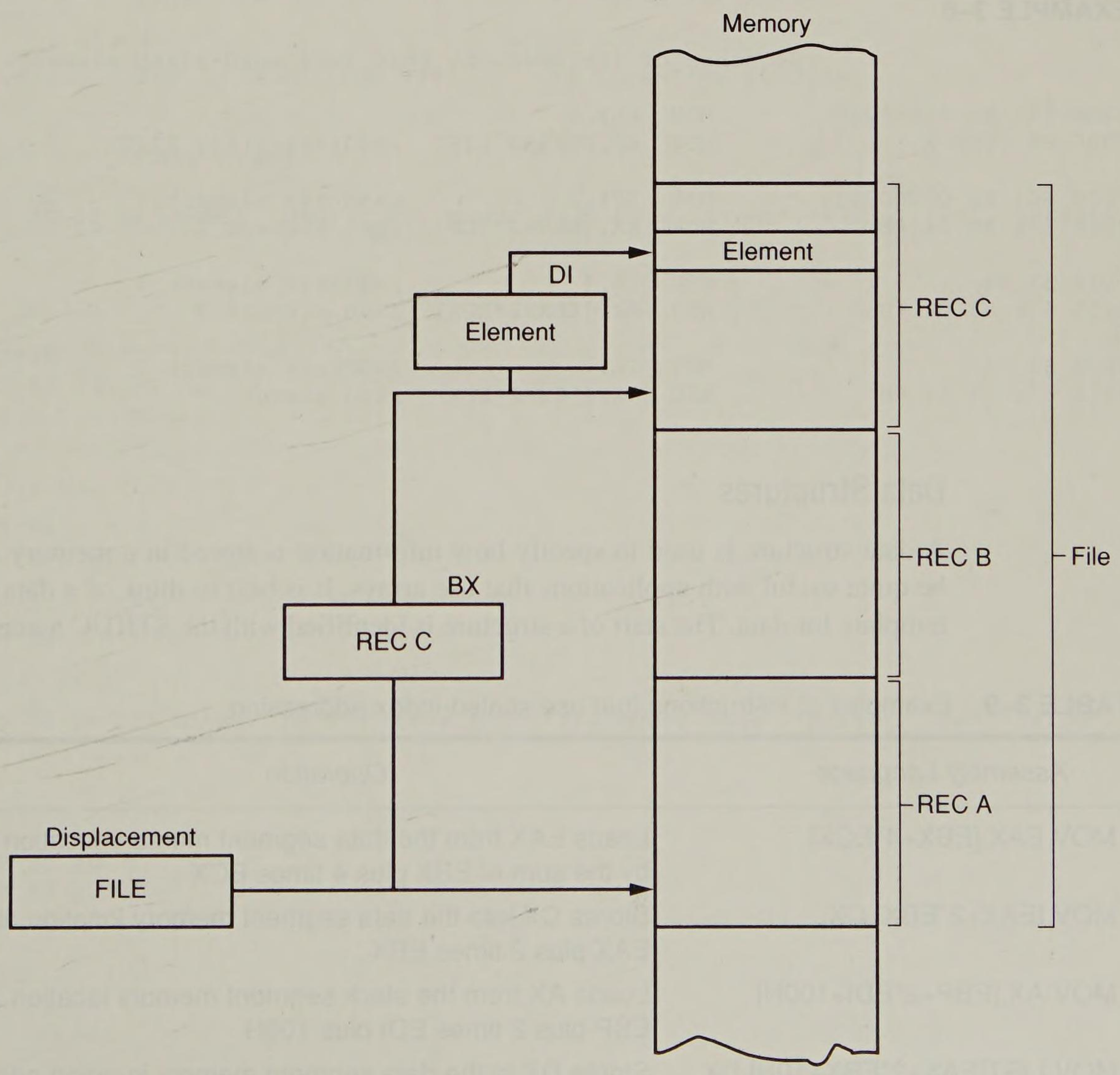



**FIGURE 3–13** Base relative-plus-index addressing used to address a FILE that contains multiple records (REC); each record contains many elements.

to address word-sized memory arrays, a scaling factor of 4× is used with doubleword-sized memory arrays, and a scaling factor of 8× is used with quadword-sized memory arrays.

An example instruction is MOV AX,[EDI+2*ECX]. This instruction uses a scaling factor of 2×, which multiplies the contents of ECX by 2 before adding it to the EDI register to form the memory address. If ECX contains a 00000000H, memory element 0 is addressed; if ECX contains a 00000001H, memory element 1 is accessed and so forth. This scales the index (ECX) by a factor of 2 for a word-sized memory array. Refer to Table 3–9 for some examples of scaled-index addressing. As you can imagine, there are an extremely large number of the scaled-index addressed register combinations.

Example 3–6 shows a sequence of instructions that uses scaled-index addressing to access a word-sized array of data called LIST. Note that the offset address of LIST is loaded into register EBX with the MOV EBX,OFFSET LIST instruction. Once EBX addresses array LIST, the elements (located in ECX) of 2, 4, and 7 of this word-wide array are added using a scaling factor of 2 to access the elements.

**EXAMPLE 3–6**

```
                        ;Sequence of instructions that sums word-sized elements of LIST
                        ;
0000 66| BB 00000000          MOV  EBX,0
0006 BB 0000 R                MOV  BX,OFFSET LIST  ;address array LIST

0009 66| B9 00000002          MOV  ECX,2           ;address element 2
000F 67& 8B 04 4B             MOV  AX,[EBX+2*ECX]  ;get element 2

0013 B1 04                    MOV  CL,4            ;address element 4
0015 67& 03 04 4B             ADD  AX,[EBX+2*ECX]  ;add element 4

0019 B1 07                    MOV  CL,7            ;address element 7
001B 67& 03 04 4B             ADD  AX,[EBX+2*ECX]  ;add element 7
```

## Data Structures

A data structure is used to specify how information is stored in a memory array and can be quite useful with applications that use arrays. It is best to think of a data structure as a template for data. The start of a structure is identified with the STRUC assembly language

**TABLE 3–9** Examples of instructions that use scaled-index addressing

| *Assembly Language* | *Operation* |
|---|---|
| MOV EAX,[EBX+4*ECX] | Loads EAX from the data segment memory location addressed by the sum of EBX plus 4 times ECX |
| MOV [EAX+2*EBX],CX | Stores CX into the data segment memory location addressed by EAX plus 2 times EBX |
| MOV AX,[EBP+2*EDI+100H] | Loads AX from the stack segment memory location addressed by EBP plus 2 times EDI plus 100H |
| MOV LIST[EAX+2*EBX+10H],DX | Stores DX in the data segment memory location addressed by LIST plus EAX plus 2 times EBX plus 10H |

directive, and the end of a structure, with the ENDS statement. A typical data structure is defined and used three times in Example 3–7. Notice that the name of the structure appears with the STRUC statement and also with the ENDS statement.

**EXAMPLE 3–7**

```
                        ;Define INFO data structure
                        ;
0057                    INFO    STRUC
0000 0020 [             NAMES   DB    32 DUP (?)        ;32 bytes for name
             00
                ]
0020 0020 [             STREET DB     32 DUP (?)        ;32 bytes for street
             00
                ]
0040 0010 [             CITY    DB    16 DUP (?)        ;16 bytes for city
             00
                ]
0050 0002 [             STATE   DB    2 DUP (?)         ;2 bytes for state
             00
                ]
0052 0005 [             ZIP     DB    5 DUP (?)         ;5 bytes for zipcode
             00
                ]
                        INFO    ENDS

0000 42 6F 62 20 53 6D  NAME1   INFO <'Bob Smith','123 Main Street','Wanda','OH','44444'>
  69 74 68
  0017 [
          00
             ]
  31 32 33 20 4D
  61 69 6E 20 53 74
  72 65 65 74
  0011 [
          00
             ]
  57 61 6E 64 61
  000B [
          00
             ]
  4F 48 34 34 34
  34 34

0057 53 74 65 76 65 20  NAME2   INFO <'Steve Doe','222 Mouse Lane','Miller','PA','18100'>
  44 6F 65
  0017 [
          00
             ]
  32 32 32 20 4D
  6F 75 73 65 20 4C
  61 6E 65
  0012 [
          00
             ]
  4D 69 6C 6C 65
  72
  000A [
          00
             ]
```

```
     50 41 31 38 31
     30 30
00AE 42 65 6E 20 44 6F  NAME3  INFO <'Ben Dover','303 Main Street','Orender','CA','90000'>
  76 65 72
  0017 [
         00
            ]
  33 30 33 20 4D
  61 69 6E 20 53 74
  72 65 65 74
  0011 [
         00
            ]
4F 72 65 6E 64
65 72
0009 [
         00
            ]
43 41 39 30 30
30 30
```

The data structure in Example 3–7 defines five fields of information. The first is 32 bytes in length and holds a name, the second is 32 bytes in length and holds a street address, the third is 16 bytes in length for the city, the forth is 2 bytes for the state, and the fifth is 5 bytes for the ZIP code. Once the structure is defined (INFO), it can be filled as illustrated with names and addresses. Three examples of use for INFO are illustrated. Carefully note that literals are surrounded with apostrophes and the entire field is surrounded with < > symbols.

When data are addressed in a structure, we use the structure name and the field name to select a field. For example, to address the STREET in NAME2 we use the operand NAME2.STREET, where the name of the structure is first followed by a period and then the name of the field. Likewise we can use NAME3.CITY to refer to the city in structure NAME3.

**EXAMPLE 3–8**

```
                        ;Clear names in array NAME1

0000 B9 0020                    MOV  CX,32
0003 B0 00                      MOV  AL,0
0005 BE 0000 R                  MOV  SI,OFFSET NAME1.NAMES
0008 F3/AA                      REP  STOSB

                        ;Clear street in array NAME2

000A B9 0020                    MOV  CX,32
000D B0 00                      MOV  AL,0
0010 BE 0077 R                  MOV  SI,OFFSET NAME2.STREET
0013 F3/AA                      REP  STOSB

                        ;Clear zipcode in array NAME3

0015 B9 0005                    MOV  CX,5
0018 B0 00                      MOV  AL,0
001A BE 0100 R                  MOV  SI,OFFSET NAME3.ZIP
001D F3/AA                      REP  STOSB
```

A short sequence of instructions appears in Example 3–8 that clears the name field in structure NAME1, the address field of structure NAME2, and the ZIP code field of structure NAME3. The function and operation of the instructions in this program are defined in later chapters in the text. You may wish to refer back to this example once these instructions are learned.

## 3–2 PROGRAM MEMORY-ADDRESSING MODES

Program memory-addressing modes, used with the JMP and CALL instructions, consist of three distinct forms: direct, relative, and indirect. This section introduces these three addressing forms, using the JMP instruction to illustrate their operation.

### Direct Program Memory Addressing

Direct program memory addressing is what most early microprocessors used for all jumps and calls. Direct program memory addressing is also used in high-level languages as the BASIC language GOTO and GOSUB instructions. The microprocessor uses this form of addressing, but doesn't use it nearly as often as relative and indirect program memory addressing.

The instructions for direct program memory addressing store the address with the opcode. For example, if a program jumps to memory location 10000H for the next instruction, the address (10000H) is stored following the opcode in the memory. Figure 3–14 shows the direct intersegment JMP instruction and the four bytes required to store the address 10000H. This JMP instruction loads CS with 1000H and IP with 0000H to jump to memory location 10000H for the next instruction. (An *intersegment* jump is a jump to any memory location within the memory system.) We often call the direct jump a *far* jump because it can jump to any memory location for the next instruction. In the real mode, a far jump accesses any location within the first 1M byte of memory by changing both CS and IP. In protected-mode operation, the far jump accesses a new code segment descriptor from the descriptor table, allowing it to jump to any memory location above the first 1M byte in 80286–80486 and Pentium microprocessors.

The only other instruction that uses direct program addressing is the intersegment CALL instruction. Usually the name of a memory address, called a *label*, refers to the location that is called or jumped to rather than the actual numeric address. When using a label with the CALL or JMP instruction, most assemblers select the best form of program addressing.

### Relative Program Memory Addressing

Relative program memory addressing is not available in many early microprocessors, but it is available to this family of microprocessors. The term *relative* means "relative

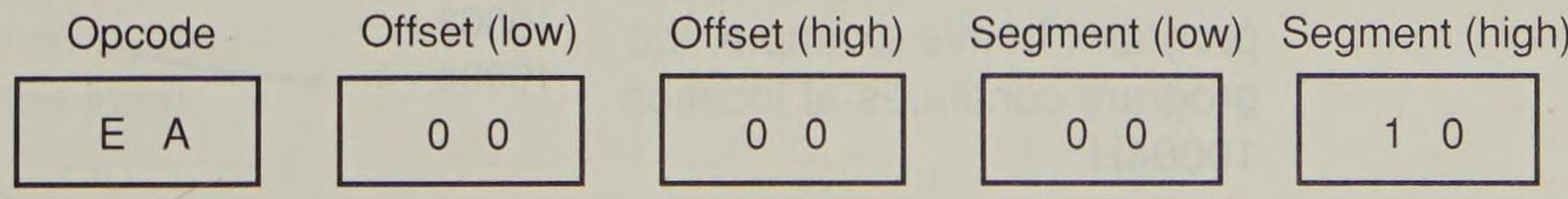



**FIGURE 3–14** The 5-byte machine language version of a JMP to memory location 10000H instruction.

to the instruction pointer (IP)." For example, if a JMP instruction skips the next two bytes of memory, the address in relation to the instruction pointer is a 2 that adds to the instruction pointer. This develops the address of the next program instruction. An example of the relative JMP instruction is shown in Figure 3–15. Notice that the JMP instruction is a one-byte instruction with a one-byte or two-byte displacement that adds to the instruction pointer. A one-byte displacement is used in *short* jumps and a two-byte displacement is used with *near* jumps and calls. Both types are considered intrasegment jumps. (An *intrasegment* jump is a jump anywhere within the current code segment.) In the 80386/80486/Pentium microprocessor the displacement can also be a 32-bit value allowing them to use relative addressing to any location within its up-to-4G byte code segment.

Relative JMP and CALL instructions contain either an 8-bit or a 16-bit signed displacement that allows a forward memory reference or a reverse memory reference. (The 80386/80486/Pentium can have an 8-bit or 32-bit displacement.) All assemblers automatically calculate the distance for the displacement and select the proper one-, two- or, four-byte form. If the distance is too far for a two-byte displacement in an 8086–80286 microprocessor, some assemblers use the direct jump. An 8-bit displacement (*short*) has a jump range of between +127 and –128 byte from the next instruction, while a 16-bit displacement (*near*) has a range of between ± 32K bytes. In the 80386/80486, a 32-bit displacement allows a range of ± 2G bytes.

## Indirect Program Memory Addressing

The microprocessor allows several forms of program indirect memory addressing for the JMP and CALL instructions. Table 3–10 lists some acceptable program indirect jump instructions, which can use any 16-bit register (AX, BX, CX, DX, SP, BP, DI, or SI), any relative register ([BP], [BX], [DI], or [SI]), and any relative register with a displacement. In the 80386/80486, extended register can also be used to hold the address or indirect address of a relative JMP or CALL. For example, the 80386/80486 can use a JMP EAX to jump to the location address by register EAX.

If a 16-bit register holds the address of a JMP instruction, the jump is near. For example, if the BX register contains a 1000H and a JMP BX instruction executes, the microprocessor jumps to offset address 1000H in the current code segment.

If a relative register holds the address, the jump is also considered an indirect jump. For example, a JMP [BX] refers to memory location within the data segment at the offset address contained in BX. At this offset address is a 16-bit number that is used as the offset address in the intrasegment jump. We at times call this type of jump an indirect–indirect or double-indirect jump.

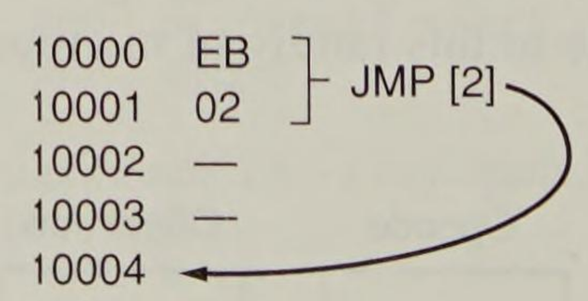



**FIGURE 3–15** A JMP [2] instruction, which will skip over the next two bytes in the program. In this example, the program continues at location 10004H.

**TABLE 3–10** Examples of program indirect addressing

| *Assembly Language* | *Operation* |
|---|---|
| JMP AX | Jump to the location addressed by AX in the current code segment |
| JMP CX | Jump to the location addressed by CX in the current code segment |
| JMP NEAR PTR [BX] | Jump to the current code segment location using the address stored at the data segment location plus BX |
| JMP NEAR PTR [DI+2] | Jump to the current code segment location using the address stored at the data segment location plus DI + 2 |
| JMP TABLE [BX] | Jump to the current code segment location addressed by the contents of TABLE plus BX |
| JMP ECX | Jump to the current code segment location addressed by ECX |

Figure 3–16 shows a jump table that is stored beginning at memory location TABLE. This jump table is referenced by the short program of Example 3–9. In this example, the BX register is loaded with a 4 so, when it combines in the JMP TABLE[BX] instruction with TABLE, the effective address is the contents of the second entry in the jump table.

**EXAMPLE 3–9**

```
                        ;Using indirect addressing for a jump
                        ;
0000 BB 0004                    MOV  BX,4              ;address LOC2
0003 FF A7 23A1 R               JMP  TABLE[BX]         ;jump to LOC2
```

## 3–3 STACK MEMORY ADDRESSING

The stack plays an important role in all microprocessors. It holds data temporarily and stores return addresses for procedures. The stack memory is a LIFO (last-in, first-out) memory, which describes the way that data are stored and removed from the stack. Data are placed onto the stack with a PUSH instruction and removed with a POP instruction.

```
TABLE  DW  LOC0
       DW  LOC1
       DW  LOC2
       DW  LOC3
```

**FIGURE 3–16** A jump table that stores addresses of various programs. The exact address chosen from the TABLE is determined by an index stored with the jump instruction.

The CALL instruction also uses the stack to hold the return address for procedures and a RET (return) instruction to remove the return address from the stack.

The stack memory is maintained by two registers: the *stack pointer* (SP or ESP) and the *stack segment register* (SS). Whenever a word of data is pushed onto the stack (see Figure 3–17(a)), the high-order 8 bits are placed in the location addressed by SP–1. The low-order 8 bits are placed in the location addressed by SP–2. The SP is then decremented by 2 so the next word of data is stored in the next available stack memory location. The SP/ESP register always points to an area of memory located within the stack segment. The SP/ESP register adds to SS × 10H to form the stack memory address in the real mode. In protected-mode operation, the SS register holds a selector that accesses a descriptor for the base address of the stack segment.

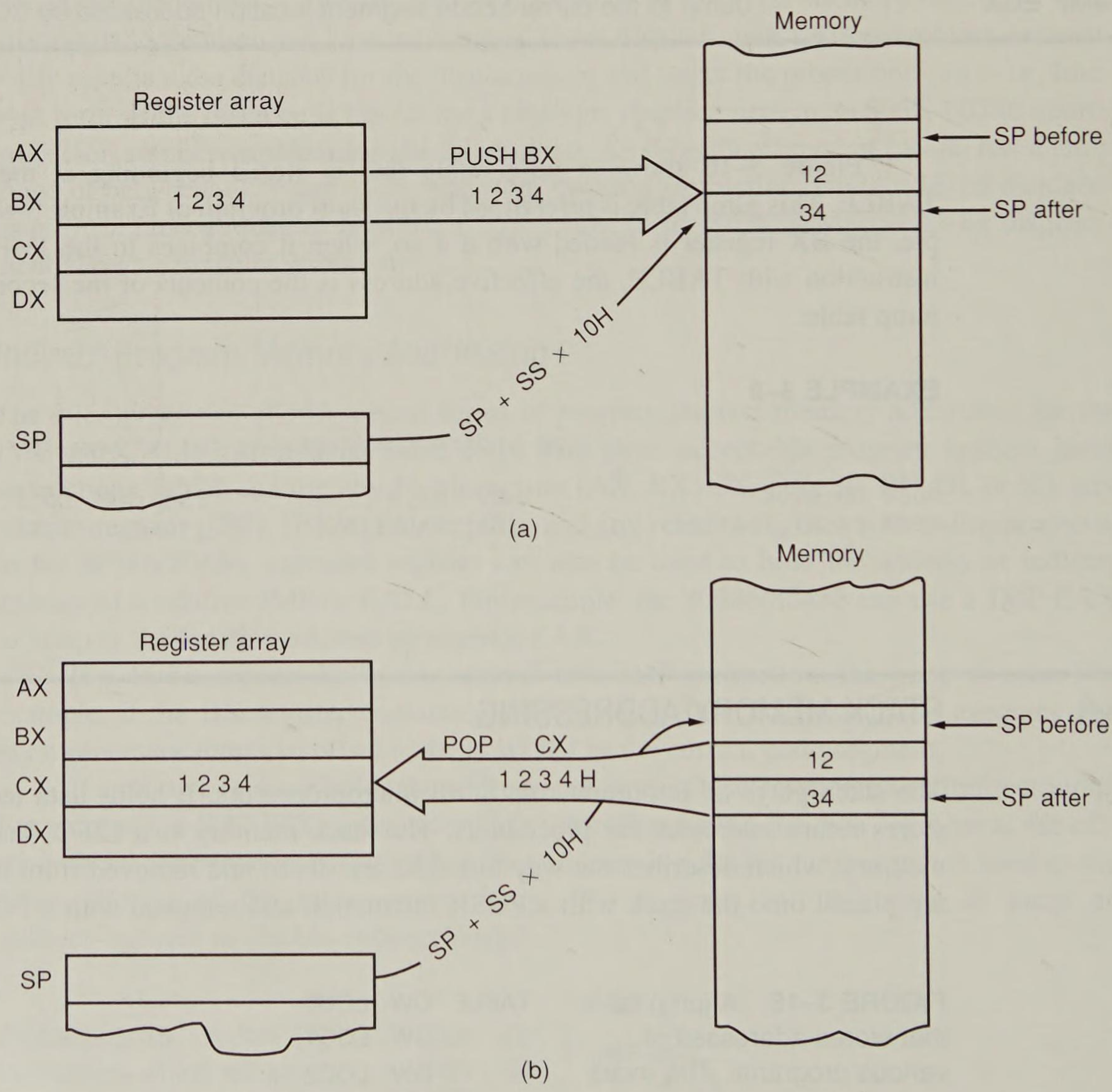



**FIGURE 3–17** The PUSH and POP instructions. (a) PUSH BX places the contents of the BX register onto the stack addressed by the SP + SS × 10H. (b) POP CX removes data from the stack at the location addressed by SP + SS × 10H and places the data into the CX register.

Whenever data are popped from the stack (see Figure 3–17(b)), the low-order 8 bits are removed from the location addressed by SP. The high-order 8 bits are removed from the location addressed by SP + 1. The SP register is then incremented by 2. Table 2–10 lists some of the PUSH and POP instructions available to the microprocessor. Note that PUSH and POP always store or retrieve *words* of data—never bytes—in the 8086–80286 microprocessor. The 80386/80486/Pentium allows words or doublewords to be transferred to and from the stack. Data may be pushed onto the stack from any 16-bit register or segment register and in the 80386/80486/Pentium any 32-bit extended register. Data may be popped off the stack into any 16-bit register or any segment register except CS. The reason that we may not pop data from the stack into CS is that this only changes part of the address of the next instruction.

The PUSHA and POPA instructions either push or pop all of the registers, except the segment registers, on the stack. These instructions are not available on the early 8086/8088 microprocessor. The push immediate instruction is also new to the 80286–80486 and Pentium microprocessors. Note the examples in Table 3–11 that show the order of the registers transferred by the PUSHA and POPA instructions. The 80386/80486/Pentium also allows extended register to be pushed or popped.

**TABLE 3–11** Example PUSH and POP instructions

| *Assembly Language* | *Operation* |
|---|---|
| POPF | Removes a word from the stack and places it into the flags |
| POPFD | Removes a doubleword from the stack and places it into the EFLAGS |
| PUSHF | Stores a copy of the flag word on the stack |
| PUSHFD | Stores a copy of the EFLAGS on the stack |
| PUSH AX | Stores a copy of AX on the stack |
| POP BX | Removes a word from the stack and places it into BX |
| PUSH DS | Stores a copy of DS on the stack |
| PUSH 1234H | Stores a 1234H on the stack |
| POP CS | Illegal instruction |
| PUSH [BX] | Stores a copy of the word contents of the memory location addressed by BX in the data segment on the stack |
| PUSHA | Stores a copy of registers AX, CX, DX, BX, SP, BP, DI, and SI on the stack |
| PUSHAD | Stores a copy of registers EAX, ECX, EDX, EBX, ESP, EBP, EDI, and ESI on the stack |
| POPA | Removes data from the stack and places it into SI, DI, BP, SP, BX, DX, CX, and AX |
| POPAD | Removes data from the stack and places it into ESI, EDI, EBP, ESP, EBX, EDX, ECX, and EAX |
| POP EAX | Removes data from the stack and places into EAX |
| PUSH EDI | Stores a copy of EDI onto the stack |

## 3–4 SUMMARY

1. The data-addressing modes include: register, immediate, direct, register indirect, base-plus-index, register relative, and base relative-plus-index addressing. In the 80386/80486/Pentium microprocessor an additional addressing mode, called scaled-index addressing, exists.
2. The program memory-addressing modes include: direct, relative, and indirect addressing.
3. Table 3–12 lists all real-mode data-addressing modes available to the 8086–80286 microprocessor. In the protected mode, the function of the segment register is to address a descriptor that contains the base address of the memory segment.
4. The 80386, 80486, and Pentium microprocessors have additional addressing modes that allow the extended registers EAX, EBX, ECX, EDX, EBP, EDI, and ESI to address memory. These addressing modes are too numerous to list in tabular form, but, in general, any of these registers function in the same way as those listed in Table 3–12. For example, the MOV AL,TABLE[EBX+2*ECX+10H] is a valid addressing mode for the 80386/80486/Pentium microprocessor.
5. The MOV instruction copies the contents of the source operand into the destination operand. The source never changes for any instruction.
6. Register addressing specifies any 8-bit register (AH, AL, BH, BL, CH, CL, DH, or DL) or any 16-bit register (AX, BX, CX, DX, SP, BP, SI, or DI). The segment registers (CS, DS, ES, or SS) are also addressable for moving data between a segment register and a 16-bit register/memory location or for PUSH and POP. In the 80386/80486/Pentium microprocessor, the extended registers also are used for register addressing and consist of: EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI. Also available to the 80386/80486 are the FS and GS segment registers.
7. The MOV immediate instruction transfers the byte or word immediately following the opcode into a register or a memory location. Immediate addressing manipulates constant data in a program. In the 80386/80486/Pentium a doubleword immediate data may also be loaded into a 32-bit register or memory location.
8. Direct addressing occurs in two forms in the microprocessor: (1) direct addressing and (2) displacement addressing. Both forms of addressing are identical except direct addressing is used to transfer data between EAX, AX, or AL and memory while displacement addressing is used with any register–memory transfer. Direct addressing requires three bytes of memory, while displacement addressing requires four bytes. Note that some of these instructions in the 80386/80486 and Pentium may require additional bytes in the form of prefixes for register and operand sizes.
9. Register indirect addressing allows data to be addressed at the memory location pointed to by either a base (BP and BX) or index register (DI and SI). In the 80386/80486/Pentium, extended register EAX, EBX, ECX, EDX, EBP, EDI, and ESI are used to address memory data.
10. Base-plus-index addressing often addresses data in an array. The memory address for this mode is formed by adding a base register, index register, and the contents of a segment register times 10H. In the 80386/80486/Pentium the base and index registers may be any 32-bit register except EIP and ESP.

**TABLE 3–12** 8086–80286 real-mode data-addressing modes

| *Assembly Language* | *Address Generation* |
|---|---|
| MOV AL,BL | 8-bit register addressing |
| MOV DI,BP | 16-bit register addressing |
| MOV DS,BX | Segment register addressing |
| MOV AL,LIST | (DS × 10H) + LIST |
| MOV CH,DATA1 | (DS × 10H) + DATA1 |
| MOV DS,DATA2 | (DS × 10H) + DATA2 |
| MOV AL,12 | Immediate data of 12 decimal |
| MOV AL,[BP] | (SS × 10H) + BP |
| MOV AL,[BX] | (DS × 10H) + BX |
| MOV AL,[DI] | (DS × 10H) + DI |
| MOV AL,[SI] | (DS × 10H) + SI |
| MOV AL,[BP+2] | (SS × 10H) + BP + 2 |
| MOV AL,[BX–4] | (DS × 10H) + BX – 4 |
| MOV AL,[DI+1000H] | (DS × 10H) + DI + 1000H |
| MOV AL,[SI+300H] | (DS × 10H) + SI + 0300H |
| MOV AL,LIST[BP] | (SS × 10H) + BP + LIST |
| MOV AL,LIST[BX] | (DS × 10H) + BX + LIST |
| MOV AL,LIST[DI] | (DS × 10H) + DI + LIST |
| MOV AL,LIST[SI] | (DS × 10H) + SI + LIST |
| MOV AL,LIST[BP+2] | (SS × 10H) + BP + LIST + 2 |
| MOV AL,LIST[BX–6] | (DS × 10H) + BX + LIST – 6 |
| MOV AL,LIST[DI+100H] | (DS × 10H) + DI + LIST + 100H |
| MOV AL,LIST[SI+20H] | (DS × 10H) + SI + LIST + 20H |
| MOV AL,[BP+DI] | (SS × 10H) + BP + DI |
| MOV AL,[BP+SI] | (SS × 10H) + BP + SI |
| MOV AL,[BX+DI] | (DS × 10H) + BX + DI |
| MOV AL,[BX+SI] | (DS × 10H) + BX + SI |
| MOV AL,[BP+DI+2] | (SS × 10H) + BP + DI + 2 |
| MOV AL,[BP+SI–4] | (SS × 10H) + BP + SI – 4 |
| MOV AL,[BX+DI+30H] | (DS × 10H) + BX + DI + 30H |
| MOV AL,[BX+SI+10H] | (DS × 10H) + BX + SI + 10H |
| MOV AL,LIST[BP+DI] | (SS × 10H) + BP + DI + LIST |
| MOV AL,LIST[BP+SI] | (SS × 10H) + BP + SI + LIST |
| MOV AL,LIST[BX+DI] | (DS × 10H) + BX + DI + LIST |
| MOV AL,LIST[BX+SI] | (DS × 10H) + BX + SI + LIST |
| MOV AL,LIST[BP+DI+2] | (SS × 10H) + BP + DI + LIST + 2 |
| MOV AL,LIST[BP+SI–7] | (SS × 10H) + BP + SI + LIST – 7 |
| MOV AL,LIST[BX+DI–10H] | (DS × 10H) + BX + DI + LIST – 10H |
| MOV AL,LIST[BX+SI+1AFH] | (DS × 10H) + BX + SI + LIST + 1AFH |

11. Register relative addressing uses either a base or index register plus a displacement to access memory data.
12. Base relative-plus-index addressing is useful for addressing a two-dimensional memory array. The address is formed by adding a base register, an index register, displacement, and the contents of a segment register times 10H.
13. Scaled-index addressing is unique to the 80386/80486/Pentium microprocessor. The second of two registers (index) is scaled by a factor of 2×, 4×, or 8× to access words, doublewords, or quadwords in memory arrays.
14. Data structures are templates for storing arrays of data and are addressed by array name and field. For example, array NUMBER and field TEN of array NUMBER is addressed as NUMBER.TEN.
15. Direct program memory addressing is allowed with the JMP and CALL instructions to any location in the memory system. With this addressing mode, the offset address and segment address are stored with the instruction.
16. Relative program addressing allows a JMP or CALL instruction to branch forward or backward in the current code segment by ± 32K bytes. In the 80386/80486/Pentium the 32-bit displacement allows a branch to any location in the current code segment using a displacement value of ± 2G bytes.
17. Indirect program addressing allows the JMP or CALL instructions to address another portion of the program or subroutine indirectly through a register or memory location.
18. The PUSH and POP instructions transfer a word between the stack and a register or memory location. A PUSH immediate instruction is available to place immediate data on the stack. The PUSHA and POPA instructions transfer AX, CX, DX, BX, BP, SP, SI, and DI between the stack and these registers. In the 80386/80486/Pentium, the extended register and extended flags can also be transferred between registers and the stack. A PUSHFD stores the EFLAGS, while a PUSHF stores the FLAGS.

## 3–5 QUESTIONS AND PROBLEMS

1. What do the following MOV instructions accomplish?
   (a) MOV AX,BX
   (b) MOV BX,AX
   (c) MOV BL,CH
   (d) MOV ESP,EBP
   (e) MOV AX,CS
2. List the 8-bit registers that are used for register addressing.
3. List the 16-bit registers that are used for register addressing.
4. List the 32-bit registers that are used for register addressing in the 80386, 80486, and Pentium microprocessors.
5. List the 16-bit segment registers used with register addressing by MOV, PUSH, and POP.

6. What is wrong with the MOV BL,CX instruction?
7. What is wrong with the MOV DS,SS instruction?
8. Select an instruction for each of the following tasks:
   (a) copy EBX into EDX
   (b) copy BL into CL
   (c) copy SI into BX
   (d) copy DS into AX
   (e) copy AL into AH
9. Select an instruction for each of the following tasks:
   (a) move a 12H into AL
   (b) move a 123AH into AX
   (c) move a 0CDH into CL
   (d) move a 1000H into SI
   (e) move a 1200A2H into EBX
10. What special symbol is sometimes used to denote immediate data?
11. What is a displacement? How does it determine the memory address in a MOV [2000H],AL instruction?
12. What do the symbols [ ] indicate?
13. Suppose that DS = 0200H, BX = 0300H, and DI = 400H. Determine the memory address accessed by each of the following instructions assuming real-mode operation:
   (a) MOV AL,[1234H]
   (b) MOV EAX,[BX]
   (c) MOV [DI],AL
14. What is wrong with a MOV [BX],[DI] instruction?
15. Choose an instruction that requires BYTE PTR.
16. Choose an instruction that requires WORD PTR.
17. Choose an instruction that requires DWORD PTR.
18. Explain the difference between the MOV BX,DATA instruction and the MOV BX,OFFSET DATA instruction.
19. Given that DS = 1000H, SS = 2000H, BP = 1000H, and DI = 0100H, determine the memory address accessed by each of the following assuming real-mode operation:
   (a) MOV AL,[BP+DI]
   (b) MOV CX,[DI]
   (c) MOV EDX,[BP]
20. What, if anything, is wrong with a MOV AL,[BX][SI] instruction?
21. Given that DS = 1200H, BX = 0100H, and SI = 0250H, determine the address accessed by each of the following instructions assuming real-mode operation:
   (a) MOV [100H],DL
   (b) MOV [SI+100H],EAX
   (c) MOV DL,[BX+100H]
22. Given that DS = 1100H, BX = 0200H, LIST = 0250H, and SI = 0500H, determine the address accessed by each of the following instructions assuming real-mode operation:
   (a) MOV LIST[SI],EDX
   (b) MOV CL,LIST[BX+SI]
   (c) MOV CH,[BX+SI]

23. Given that DS = 1300H, SS = 1400H, BP = 1500H, and SI = 0100H, determine the address accessed by each of the following instructions assuming real-mode operation:
    (a) MOV EAX,[BP+200H]
    (b) MOV AL,[BP+SI-200H]
    (c) MOV AL,[SI-0100H]
24. Which base register addresses data in the stack segment?
25. Given that EAX = 00001000H, EBX = 00002000H and DS = 0010H determine the addresses accessed by the following instructions assuming real-mode operation:
    (a) MOV ECX,[EAX+EBX]
    (b) MOV [EAX+2*EBX],CL
    (c) MOV DH,[EBX+4*EAX+1000H]
26. Develop a data structure that will have five fields of one word each named F1, F2, F3, F4, and F5 and a structure name of FIELDS.
27. Show how field F3 of the data structure constructed in question 26 is addressed in a program.
28. List all three program memory-addressing modes.
29. How many bytes of memory store a far direct jump instruction? What is stored in each of the bytes?
30. What is the difference between an intersegment and intrasegment jump?
31. If a near jump uses a signed 16-bit displacement, how can it jump to any memory location within the current code segment?
32. The 80386/80486 and Pentium use a _____-bit displacement to jump to any location within the 4G byte code segment.
33. What is a far jump?
34. If a JMP instruction is stored at memory location 100H within the current code segment, it cannot be a _____ jump if it is jumping to memory location 200H within the current code segment.
35. Show which JMP instruction assembles (short, near, or far) if the JMP THERE instruction is stored at memory address 10000H and the address of THERE is:
    (a) 10020H
    (b) 11000H
    (c) 0FFFEH
    (d) 30000H
36. Form a JMP instruction that jumps to the address pointed to by the BX register.
37. Select a JMP instruction that jumps to the location stored in memory at location table. Assume that it is a near JMP.
38. How many bytes are stored on the stack by PUSH instructions?
39. Explain how the PUSH [DI] instruction functions.
40. Which registers are placed in the stack and in what order by the PUSHA instruction?
41. What does the 80386/80486/Pentium instruction PUSHAD accomplish?
42. Which instruction places the EFLAGS on the stack in the Pentium microprocessor?

# CHAPTER 4

# Data Movement Instructions

## INTRODUCTION

In this chapter, we explain the data movement instructions. The data movement instructions include: MOV, MOVSX, MOVZX, PUSH, POP, BSWAP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LFS, LGS, LSS, LAHF, SAHF, and the string instructions: MOVS, LODS, STOS, INS, and OUTS. We present data movement instructions before arithmetic and logic instructions or program control instructions because they are used more frequently and are more easily understood.

The microprocessor requires an assembler program, which generates machine language, because machine language instructions are too complex to be efficiently generated by hand. This chapter describes the assembly language syntax and some of its directives. (This text assumes that the user is developing software on an IBM personal computer or clone using the Microsoft MACRO assembler (MASM), Intel Assembler (ASM), Borland Turbo assembler (TASM), or similar software. This text presents information that functions with the Microsoft MASM assembler, but most programs assemble without modification with other assemblers. Appendix A explains the Microsoft assembler and provides details on the linker program and the programmer's workbench.)

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Explain the operation of each data movement instruction with applicable addressing modes.
2. Explain the purposes of the assembly language pseudo–operations and key words such as: ALIGN, ASSUME, DB, DD, DW, END, ENDS, ENDP, EQU, OFFSET, ORG, PROC, PTR, SEGMENT, USE16, USE32, and USES.
3. Select the appropriate assembly language instruction to accomplish a specific data movement task.

4. Determine the symbolic opcode, source, destination, and addressing mode for a given hexadecimal machine language instruction.
5. Use the assembler to set up a data segment, stack segment, and a code segment.
6. Show how to set up a procedure using PROC and ENDP.
7. Explain the difference between memory models and full segment definitions for the MASM assembler.

## 4–1 MOV REVISITED

We used the MOV instruction in Chapter 3 to explain the diverse 8086–80486 and Pentium addressing modes. In this chapter, we use MOV to introduce the machine language instructions available for various addressing modes and instructions. Machine code is introduced because it may occasionally be necessary to interpret machine language programs generated by an assembler. Interpretation of the machine's native language allows debugging or modification at the machine language level. Occasionally we still make machine language patches using the DEBUG program available with DOS, which requires some knowledge of machine language. We also show how to convert between machine and assembly language instructions using Appendix B.

### Machine Language

*Machine language* is the native binary code that the microprocessor understands and uses as instructions to control its operation. Machine language instructions, for the 8086–80486 and Pentium, vary in length from one to as many as 13 bytes. Although machine language appears complex, there is order to this microprocessor's machine language. There are well over 20,000 variations of machine language instructions for the 8086–80486 and Pentium microprocessors, which means there is no complete list of these variations. Because of this, some binary bits in a machine language instruction are given, and the remainder must be determined for each variation of the instruction.

Instructions for the 8086–80286 are 16-bit mode instructions that take the form found in Figure 4–1(a). The 16-bit mode instructions are compatible with the 80386, 80486, and Pentium microprocessors if they are programmed to use the 16-bit mode instruction formats. The 80386/80486/Pentium assumes that all instructions are 16-bit mode instructions when the machine is operated in the real mode. In the protected mode, the upper byte of the 80386/80486/Pentium descriptor contains the D bit that selects either the 16- or 32-bit instruction mode. The 80386, 80486, and Pentium 32-bit mode instructions are shown in Figure 4–1(b). These instructions can occur in the 16-bit instruction mode by the use of prefixes.

The first two bytes of the 32-bit instruction mode format are called override prefixes because they are not always present. The first byte modifies the size of the operand address used by the instruction, and the second byte modifies the register size. If the 80386/80486/Pentium is operating as a 16-bit instruction mode machine (real or protected mode) and a 32-bit register is used, the register size prefix (66H) is appended to the front of the instruction. If the 80386/80486/Pentium is operating in the 32-bit instruction mode

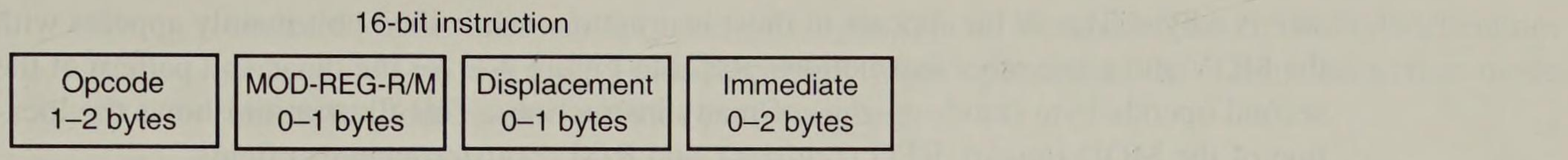


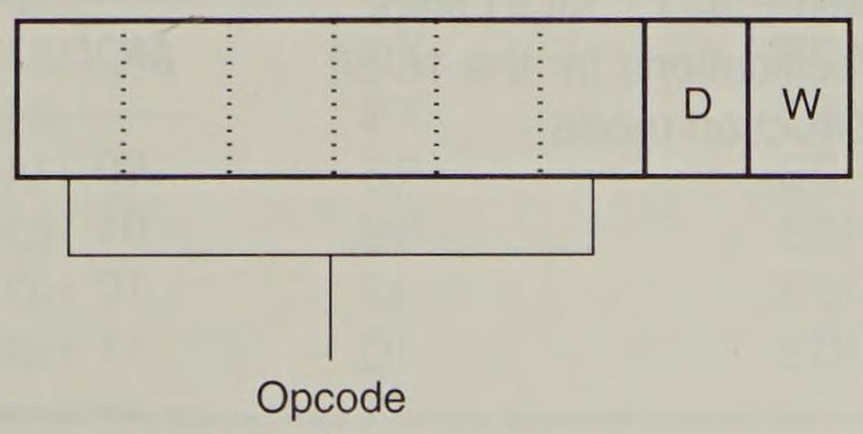

**FIGURE 4–1** The 16- and 32-bit machine language instruction formats.

and a 32-bit register is used (protected mode only), the register size prefix is not present. If a 16-bit register appears in an instruction in the 32-bit instruction mode, the register size prefix is present to select a 16-bit register. The address size prefix is used in a similar fashion as explained later. These prefixes toggle the size of the register and operand address from 16 bit to 32 bit or 32 bit to 16 bit for the prefixed instruction. Note that the 16-bit mode uses 8- and 16-bit registers and the 32-bit mode uses 8- and 32-bit registers by default. The prefixes override these defaults so a 32-bit register can be used in the 16-bit mode or a 16-bit register can be used in the 32-bit mode. The mode of operation (16 or 32 bits) should be selected to conform to the application at hand. If 8- and 32-bit data pervade the application, then the 32-bit mode should be selected; likewise, if 8- and 16-bit data pervade, then the 16-bit mode should be selected.

*The Opcode.* The *opcode* selects the operation (addition, subtraction, move, etc.) performed by the microprocessor. The opcode is either one or two bytes in length for most machine language instructions. Figure 4–2 illustrates the general form of the first opcode byte of many, but not all, machine language instructions. Here, the first six bits of the first byte are the binary opcode. The remaining two bits indicate the *direction* (D) of the data flow and whether the data are a *byte* or a *word* (W). In the 80386/80486/Pentium, words and doublewords are both specified when W = 1. The instruction mode and address size or register size prefix select whether W represents a word or a doubleword in the 80386/80486/Pentium microprocessor.

If the D bit = 1, data flow to the register (REG) field from the R/M field located in the second byte of an instruction. If the D bit = 0, data flow to the R/M field from the REG field. If the W bit = 1, the data size is a word or doubleword, and if the W bit = 0, the data

**FIGURE 4–2** The first byte of many machine language instructions, showing the position of the opcode, D, and W bits.



size is a byte. The W bit appears in most instructions while the D bit mainly appears with the MOV and some other instructions. Refer to Figure 4–3 for the binary bit pattern of the second opcode byte (mod-reg-r/m) of many instructions. This illustration shows the location of the MOD (*mode*), REG (*register*), and R/M (*register/memory*) fields.

*MOD Field.* The *MOD field* specifies the addressing mode (MOD) for the selected instruction. The MOD field selects the type of addressing and whether a displacement is present with the selected type. Table 4–1 lists the operand forms available to the MOD field for the 16-bit instruction mode in the 8086–80486 and Pentium microprocessors unless the address size override prefix appears. If the MOD field contains a 11, it selects the register addressing mode. Register addressing uses the R/M field to specify a register instead of a memory location. If the MOD field contains a 00, 01, or 10, the R/M field selects one of the data memory addressing modes. When MOD selects a data memory addressing mode, it indicates that the addressing mode contains no displacement (00), an 8-bit sign-extended displacement (01), or a 16-bit displacement (10). The MOV AL,[DI] instruction is an example showing no displacement, a MOV AL,[DI + 2] instruction uses an 8-bit displacement (+2), and a MOV AL,[DI + 1000] instruction uses a 16-bit displacement (+1000).

All 8-bit displacements are sign-extended into 16-bit displacements when the microprocessor executes the instruction. If the 8-bit displacement is 00H–7FH (positive), it is sign-extended to 0000H–007FH before adding to the offset address. If the 8-bit displacement is 80H–FFH (negative), it is sign-extended to FF80H–FFFFH. To sign-extend a number, its sign bit is copied to the next higher-order byte, which generates either a 00H or an FFH in the higher-order byte.

In the 80386/80486/Pentium microprocessor, the MOD field may be the same as Table 4–1 or, if the instruction mode is 32 bits, the MOD field may be the same as Table 4–2. The MOD field is interpreted as selected by the address size override prefix or the operating mode of the microprocessor. This change in the interpretation of the MOD field and instruction supports many of the numerous additional addressing modes allowed in the 80386/80486/Pentium microprocessor. The main difference is when the MOD field is a 10. This causes the 16-bit displacement to become a 32-bit displacement in order to allow any

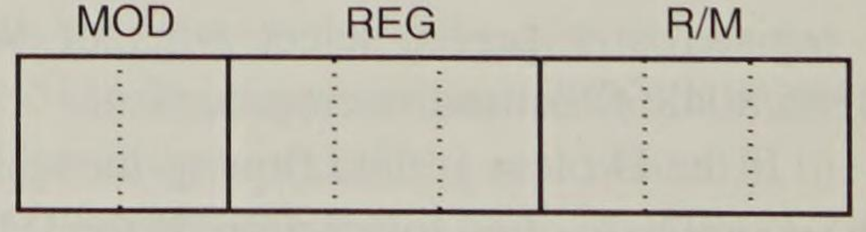



**FIGURE 4–3** The second byte of many machine language instructions, showing the position of the MOD, REG, and R/M fields.

**TABLE 4–1** MOD field specifications for the 16-bit instruction mode

| *MOD* | *Function* |
|---|---|
| 00 | No displacement |
| 01 | 8-bit sign-extended displacement |
| 10 | 16-bit displacement |
| 11 | R/M is a register |

protected-mode memory location (4G bytes) to be accessed. The 80386/80486/Pentium only allows an 8- or 32-bit displacement when operated in the 32-bit instruction mode unless the address size override prefix appears.

*Register Assignments.* Table 4–3 lists the register assignments for the REG field and the R/M field (MOD = 11). This table contains three lists of register assignments: one is used when the W bit = 0 (bytes), and the other two are used when the W bit = 1 (words or doublewords). Note that doubleword registers are only available to the 80386/80486/Pentium.

Suppose that a 2-byte instruction, 8BECH, appears in a machine language program. Because neither a 67H (address size override prefix) nor a 66H (register size override prefix) appears as the first byte, the first byte is the opcode. Assuming that the microprocessor is operated in the 16-bit instruction mode, this instruction is converted to binary and placed in the instruction format of bytes 1 and 2 as illustrated in Figure 4–4. The opcode is 100010. If you refer to Appendix B, which lists machine language instructions, you will find that this is the opcode for a MOV instruction. Also notice that both the D and W bits are a logic 1, which means that a word moves into the destination register specified in the REG field. The REG field contains a 101, indicating register BP, so the MOV instruction moves data into register BP. Because the MOD field contains a 11, the R/M field also indicates a register. Here, R/M = 100 (SP), therefore this instruction moves data from SP into BP and is written in symbolic form as a MOV BP,SP instruction.

Suppose that a 668BE8H instruction appears in an 80386/80486/Pentium operated in the 16-bit instruction mode. The first byte (66H) is the register size override prefix that selects 32-bit register operands for the 16-bit instruction mode. The remainder of the instruction indicates that the opcode is a MOV with a source operand of EAX and a destination operand of EBP. This instruction is a MOV EBP,EAX. The same instruction

**TABLE 4–2** MOD field specifications for 32-bit instruction mode (80386/80486/Pentium only)

| *MOD* | *Function* |
|---|---|
| 00 | No displacement |
| 01 | 8-bit sign-extended displacement |
| 10 | 32-bit displacement |
| 11 | R/M is a register |

**TABLE 4–3** REG and R/M assignments (when MOD = 11)

| *Code* | *W = 0 (Byte)* | *W = 1 (Word)* | *W = 1 (Doubleword)* |
|---|---|---|---|
| 000 | AL | AX | EAX |
| 001 | CL | CX | ECX |
| 010 | DL | DX | EDX |
| 011 | BL | BX | EBX |
| 100 | AH | SP | ESP |
| 101 | CH | BP | EBP |
| 110 | DH | SI | ESI |
| 111 | BH | DI | EDI |

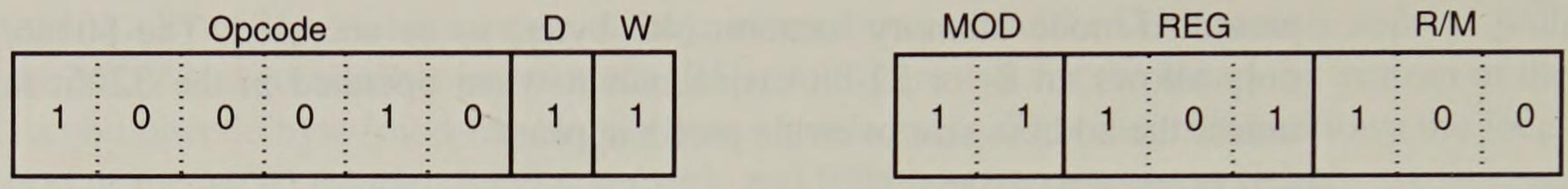



**FIGURE 4–4** The 8BEC instruction placed into the machine language formats of Figures 4–2 and 4–3. This is decoded as the MOV BP,SP instruction.

becomes a MOV BP,AX instruction in the 80386/80486/Pentium microprocessor operating in the 32-bit instruction mode, because the register size override prefix selects a 16-bit register. Luckily, the assembler program keeps track of the register and address size prefixes.

*R/M Memory Addressing.* If the MOD field contains a 00, 01, or 10, the R/M field takes on a new meaning. Table 4–4 lists the memory addressing modes for the R/M field when MOD is a 00, 01, or 10 for the 16-bit instruction mode.

All of the 16-bit addressing modes presented in Chapter 3 appear in Table 4–4. The displacement, discussed in Chapter 3, is defined by the MOD field. If MOD = 00 and R/M = 101, the addressing mode is [DI]. If MOD = 01 or 10, the addressing mode is [DI + 33H] or LIST [DI + 22H] for the 16-bit instruction mode. This example uses LIST, 33H, and 22H as arbitrary values for the displacement.

Figure 4–5 illustrates the machine language version of the 16-bit instruction MOV DL,[DI] or instruction (8A15H). This instruction is two bytes long and has an opcode 100010, D = 1 (*to REG from R/M*), W = 0 (*byte*), MOD = 00 (*no displacement*), REG = 010 (*DL*) and R/M = 101 (*[DI]*). If the instruction changes to MOV DL,[DI + 1], the MOD field changes to 01 for an 8-bit displacement, but the first two bytes of the instruction otherwise remain the same. The instruction now becomes 8A5501H instead of 8A15H. Notice that the 8-bit displacement appends to the first two bytes of the instruction to form a 3-byte

**TABLE 4–4** The 16-bit R/M memory addressing modes

| *Code* | *Addressing Mode* |
|---|---|
| 000 | DS:[BX + SI] |
| 001 | DS:[BX + DI] |
| 010 | SS:[BP + SI] |
| 011 | SS:[BP + DI] |
| 100 | DS:[SI] |
| 101 | DS:[DI] |
| 110 | SS:[BP]* |
| 111 | DS:[BX] |

**Note*: See text under Special Addressing Mode.

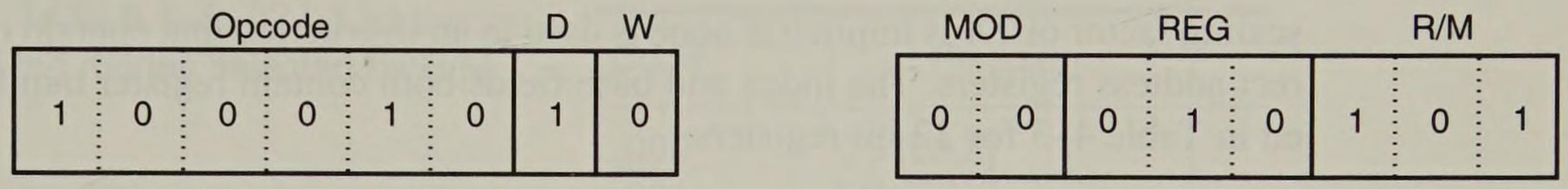



**FIGURE 4–5** A MOV DL,[DI] instruction coded in binary machine language.

instruction instead of a 2-byte of instruction. If the instruction is again changed, to MOV DL,[DI + 1000H], the machine language form becomes 8A750010H. Here the 16-bit displacement of 1000H (coded as 0010H) appends the opcode.

*Special Addressing Mode.* There is a special addressing mode that does not appear in Tables 4–2, 4–3, or 4–4. It occurs whenever memory data are referenced by only the displacement mode of addressing for 16-bit instructions. Examples are the MOV [1000H],DL and MOV NUMB,DL instructions. The first instruction moves the contents of register DL into data segment memory location 1000H. The second instruction moves register DL into symbolic data segment memory location NUMB.

Whenever an instruction has only a displacement, the MOD field is always a 00 and the R/M field is always a 110. This combination normally shows that the instruction contains no displacement and uses addressing mode [BP]. You cannot actually use addressing mode [BP] without a displacement in machine language. The assembler takes care of this by using an 8-bit displacement (MOD = 01) of 00H whenever the [BP] addressing mode appears in an instruction. This means that the [BP] addressing mode assembles as a [BP + 0] even though we use [BP].

Figure 4–6 shows the binary bit pattern required to encode the MOV [1000H],DL instruction in machine language. If the individual translating this symbolic instruction into machine language does not know about the special addressing mode, it would incorrectly translate to a MOV [BP],DL instruction. Figure 4–7 shows the actual form of the MOV [BP],DL instruction. Notice that this is a 3-byte instruction with a displacement of 00H.

*32-Bit Addressing Modes.* The 32-bit addressing modes found in the 80386/80486/Pentium microprocessor are obtained by running these machines in the 32-bit instruction mode or in the 16-bit instruction mode using the address size prefix 67H. Table 4–5 shows the coding for R/M used to specify the 32-bit addressing modes. Notice that when R/M = 100, an additional byte appears in the instruction called a *scaled index byte*. The scaled index byte indicates the additional forms of scaled index addressing that do not appear in Table 4–5. The scaled index byte is mainly used when two registers are added to specify the memory address in an instruction.

Figure 4–8 shows the format of the *scaled index byte* as selected by a value of 100 in the R/M field of an instruction when the 80386/80486/Pentium uses a 32-bit address. The left-most 2 bits select a scaling factor (multiplier) of 1×, 2×, 4×, or 8×. Note that a

scaling factor of 1X is implicit if none is used in an instruction that contain two 32-bit indirect address registers. The index and base fields both contain register numbers as indicated in Table 4–3 for 32-bit registers.

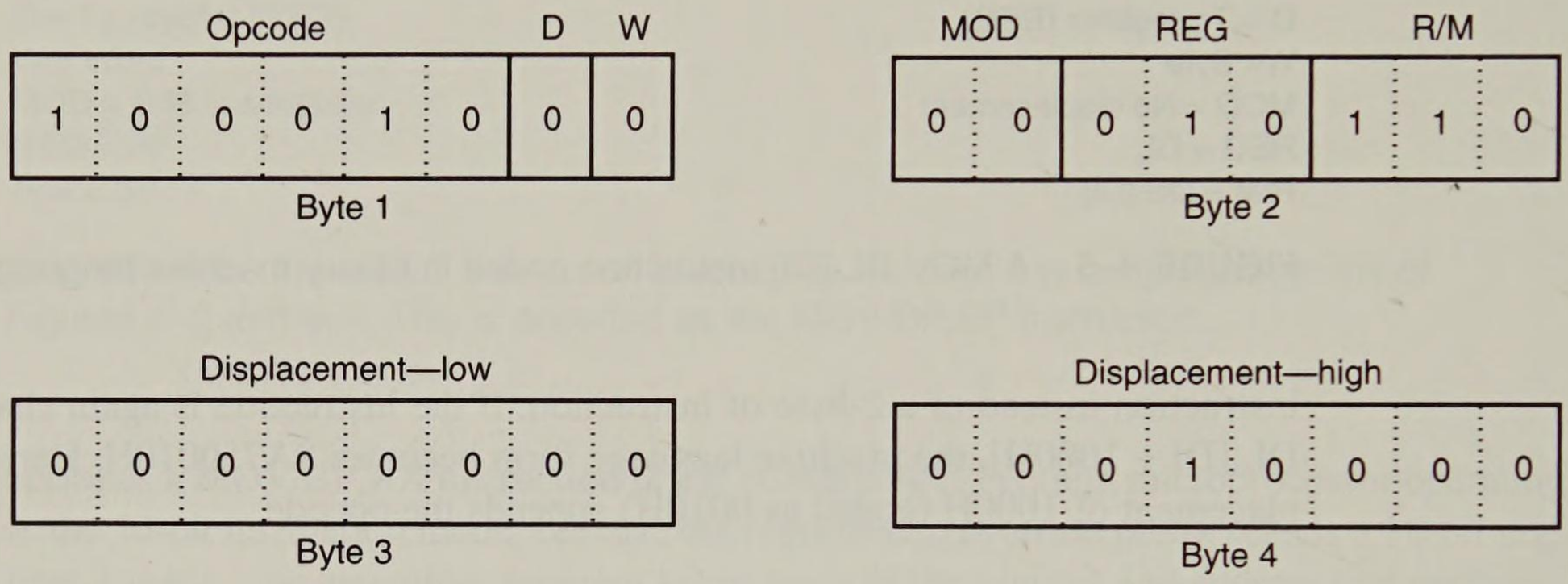



**FIGURE 4–6** The MOV [1000H],DL instruction coded into machine language. Note the two additional bytes required to store the displacement of 1000H.

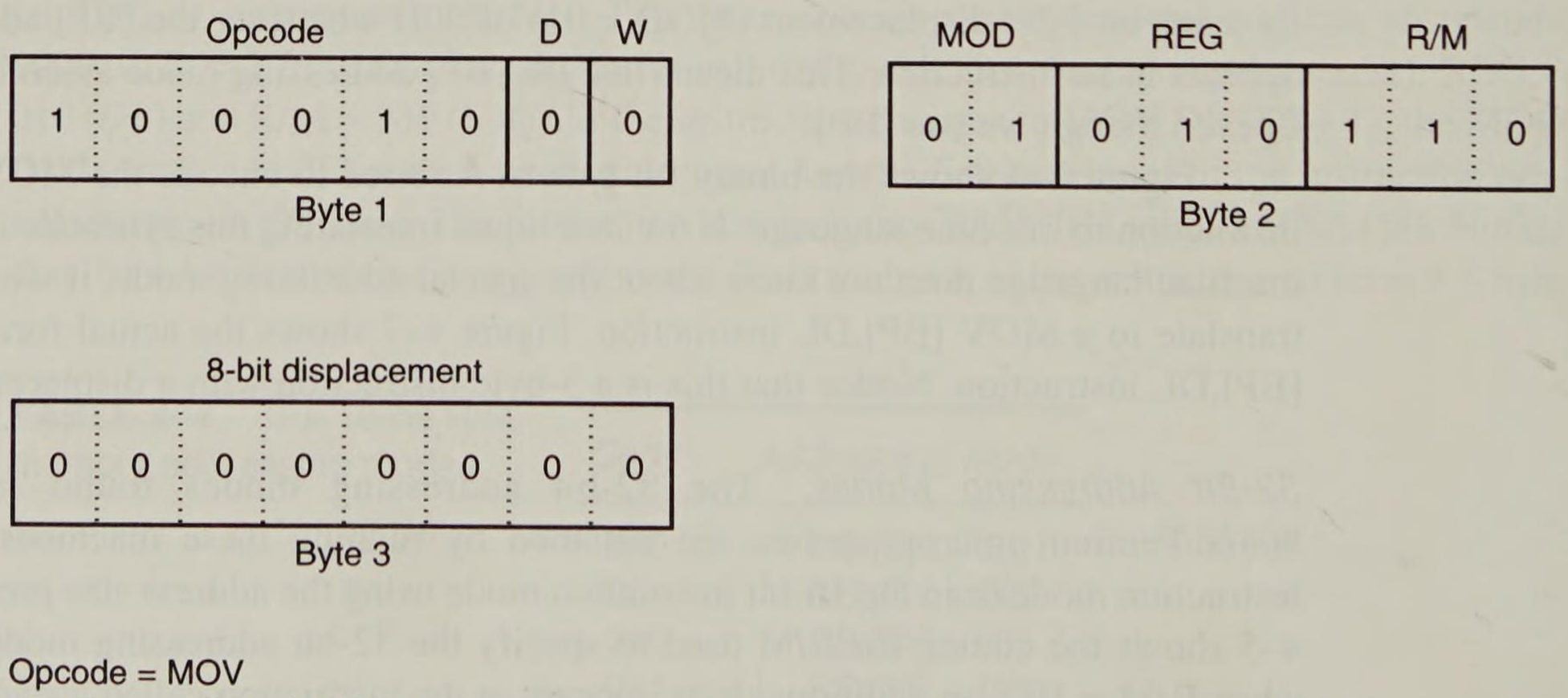


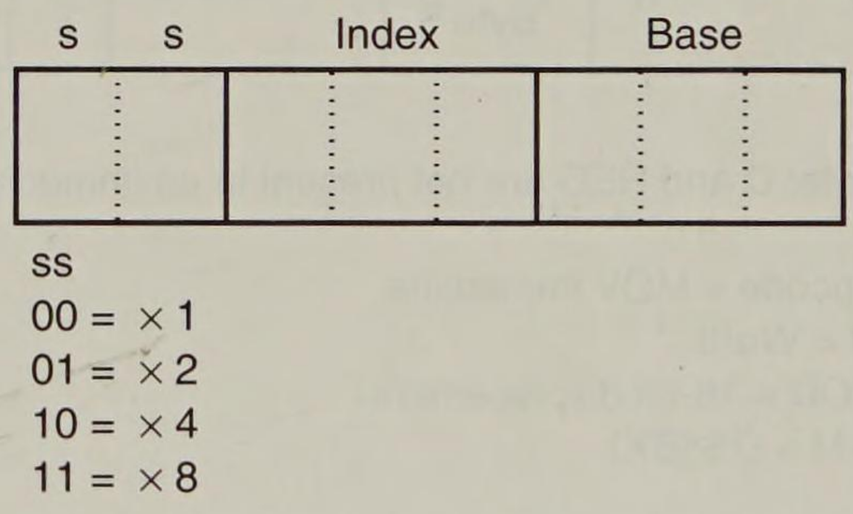

**FIGURE 4–7** The MOV [BP],DL instruction coded into machine language requires a 1-byte displacement of 00H.

**TABLE 4–5** 32-bit addressing modes selected by R/M

| *Code* | *Function* |
|---|---|
| 000 | DS:[EAX] |
| 001 | DS:[ECX] |
| 010 | DS:[EDX] |
| 011 | DS:[EBX] |
| 100 | Uses scaled index byte |
| 101 | SS:[EBP]* |
| 110 | DS:[ESI] |
| 111 | DS:[EDI] |

**Note*: If the MOD bits are 00, this addressing mode uses a 32-bit displacement without register EBP. This is similar to the special addressing mode for the 16-bit instruction mode.

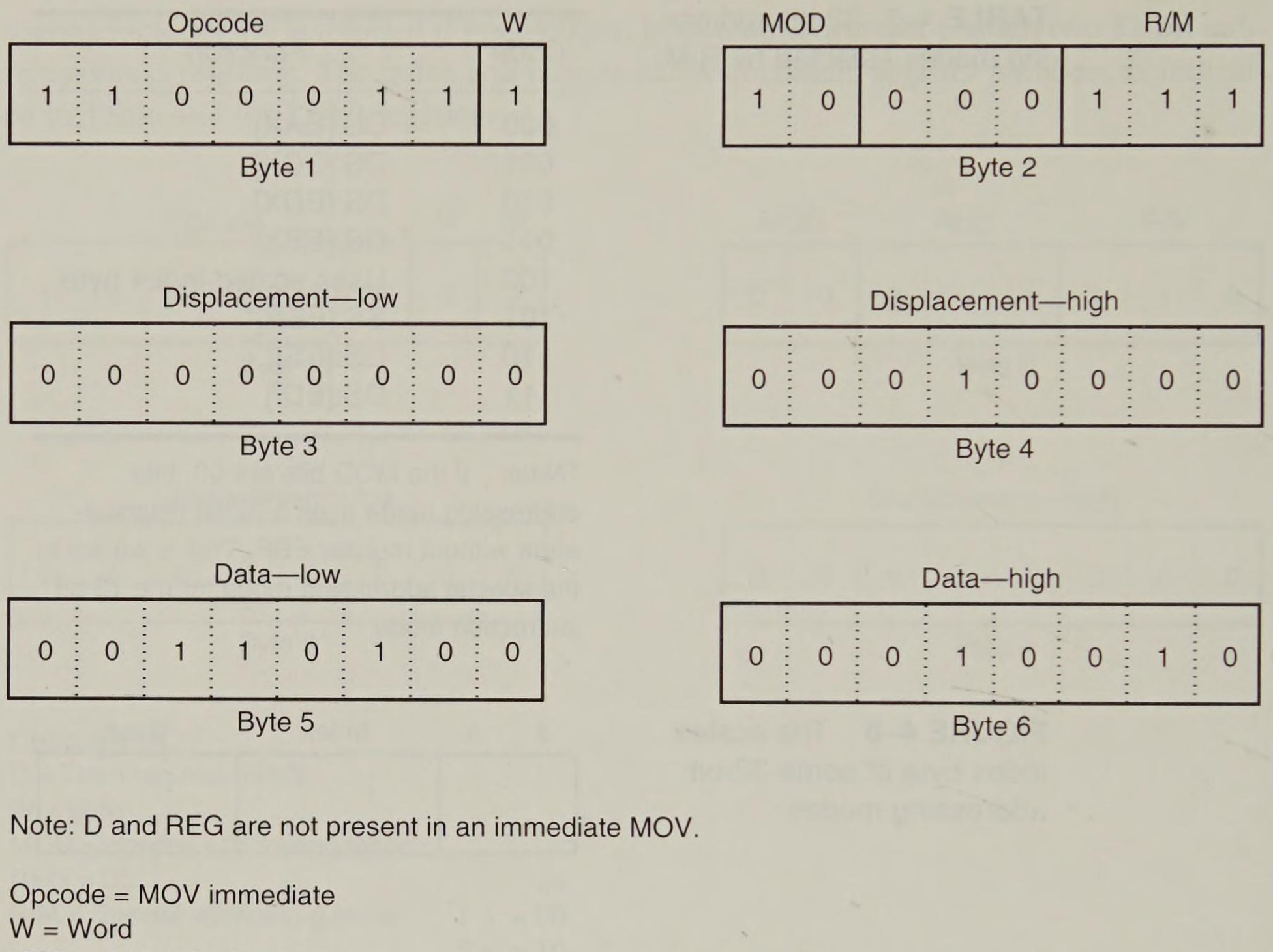
**FIGURE 4–8** The scaled index byte of some 32-bit addressing modes.




The instruction MOV EAX,[EBX+4*ECX] is encoded as 67668B048BH. Notice that both the address size (67H) and register size (66H) override prefixes appear for this instruction. This means that the instruction is 67668B048BH when operated with the 80386/80486/Pentium in the 16-bit instruction mode. If the 80386/80486/Pentium is operated in the 32-bit instruction mode both prefixes disappear so the instruction becomes 8B048BH. The use of the prefixes depends on the mode of operation of the 80386, 80486, and Pentium microprocessors.

*An Immediate Instruction.* Suppose we choose the MOV WORD PTR [BX + 1000H],1234H instruction as an example of a 16-bit instruction using immediate addressing. This instruction moves a 1234H into the word-sized memory location addressed by the sum of 1000H, BX, and DS × 10H. This 6-byte instruction uses two bytes for the opcode, W, MOD, and R/M fields. Two of the six bytes are the data of 1234H. Two of the six bytes are the displacement of 1000H. Figure 4–9 shows the binary bit pattern for each byte of this instruction.

This instruction, in symbolic form, includes WORD PTR. The WORD PTR directive indicates to the assembler that the instruction uses a word-sized memory pointer. If the instruction moves a byte of immediate data, then BYTE PTR replaces WORD PTR in the instruction. Likewise, if the instruction uses a doubleword of immediate data, the DWORD PTR directive replaces BYTE PTR. Most instructions that refer to memory




Note: D and REG are not present in an immediate MOV.

Opcode = MOV immediate
W = Word
MOD = 16-bit displacement
R/M = DS:[BX]

**FIGURE 4–9** The MOV WORD PTR [BX+1000H],1234H instruction coded into machine language. Note that this instruction is 6 bytes in length.

through a pointer do not need the BYTE PTR, WORD PTR, or DWORD PTR directives. These are only necessary when it is not clear if the operation is a byte or a word. The MOV [BX],AL instruction is clearly a byte move, while the MOV [BX],1 instruction is not exact and could therefore be a byte-, word-, or doubleword-sized move. Here the instruction must coded as MOV BYTE PTR [BX],1, MOV WORD PTR [BX],1, or MOV DWORD PTR [BX],1. If not, the assembler flags it as an error because it cannot determine the intent of this instruction.

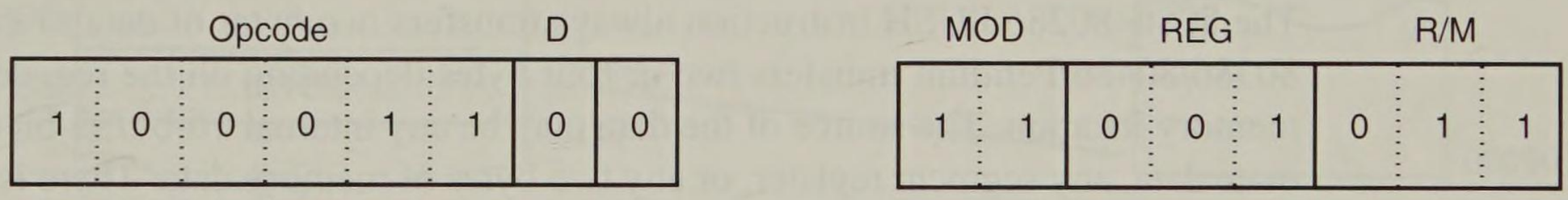
*Segment MOV Instructions.* If the contents of a segment register are moved by the MOV, PUSH, or POP instruction, a special set of register bits (REG field) selects the segment register (see Table 4–6).

Figure 4–10 shows a MOV BX,CS instruction converted to binary. The opcode for this type of MOV instruction is different from the prior MOV instructions. Segment registers can be moved between any 16-bit register or 16-bit memory location. For example, the MOV [DI],DS instruction stores the contents of DS into the memory location addressed by DI in the data segment. An immediate segment register MOV is not available in the instruction set. To load a segment register with immediate data, we must first load another register with the data and then move it to a segment register.

**TABLE 4–6** Segment register selection bits

| *Code* | *Segment Register* |
|---|---|
| 000 | ES |
| 001 | CS* |
| 010 | SS |
| 011 | DS |
| 100 | FS |
| 101 | GS |

**Note*: MOV CS,?? and POP CS are not allowed by the microprocessor. The FS and GS segments are only available to the 80386/80486/Pentium microprocessor.




Note: W is not present in this instruction.

Opcode = MOV
D = From register (REG)
MOD = R/M is a register
REG = CS
R/M = BX

**FIGURE 4–10** The MOV BX,CS instruction decoded in machine language. Here the REG field is encoded as the CS register and R/M as BX.

Although this has not been a complete coverage of machine language coding, it should give you a good start in machine language programming. Remember a program written in symbolic assembly language (assembly language), is rarely assembled by hand into binary machine language. An *assembler* converts symbolic assembly language into machine language. With over 20,000 instruction variations for the 8086–80486 microprocessor, let us hope that an assembler is available for the conversion, because the process of converting assembly language into machine language is very time-consuming, although not impossible.

## 4–2 PUSH/POP

The PUSH and POP instructions are important instructions that store and retrieve data from the LIFO (last-in, first-out) stack memory. The 8086–80486 and Pentium microprocessors have six forms of the PUSH and POP instructions: register, memory, immediate, segment register, flags, and all registers. The PUSH and POP immediate and the

PUSHA and POPA (all registers) forms are not available in the earlier 8086/8088 microprocessor, but are available to the 80286, 80386, 80486, and Pentium.

Register addressing allows the contents of any 16-bit register to be transferred to or from the stack. In the 80386/80486/Pentium, the 32-bit extended registers and flags (EFLAGS) can also be pushed or popped from the stack. Memory addressing stores the contents of a 16-bit memory location (32 bits in the 80386/80486/Pentium) on the stack or stack data into a memory location. Immediate addressing allows immediate data to be pushed onto the stack, but not popped off the stack. Segment register addressing allows the contents of any segment register to be pushed onto the stack or removed from the stack (CS may be pushed, but data from the stack may never be popped into CS). The flags may be pushed or popped from that stack and the contents of all the registers may be pushed or popped.

## PUSH

The 8086–80286 PUSH instruction always transfers *two bytes* of data to the stack and the 80386/80486/Pentium transfers two or four bytes depending on the register or size of the memory location. The source of the data may be any internal 16-bit/32-bit register, immediate data, any segment register, or any two bytes of memory data. There is also a PUSHA instruction that copies the contents of the internal register set, except the segment registers, to the stack. The PUSHA (*push all*) instruction copies the registers to the stack in the following order: AX, CX, DX, BX, SP, BP, SI, and DI. The value of SP pushed to the stack is whatever it was before the PUSHA instruction executes. The PUSHF (*push flags*) instruction copies the contents of the flag register to the stack. The PUSHAD and POPAD instructions push and pop the contents of the 32-bit register set found in the 80386, 80486, and Pentium microprocessors.

Whenever data are pushed onto the stack, the first (most significant) data byte moves into the stack segment memory location addressed by SP – 1. The second (least significant) data byte moves into the stack segment memory location addressed by SP – 2. After the data are stored by a PUSH, the contents of the SP register decrement by 2. The same is true for a doubleword push, except four bytes are moved to the stack memory (most significant byte first) then the stack pointer decrements by 4. Figure 4–11 shows the operation of the PUSH AX instruction. This instruction copies the contents of AX onto the stack where address SS:[SP – 1] = AH, SS:[SP – 2] = AL, and afterwards SP = SP – 2.

The PUSHA instruction pushes all the internal 16-bit registers onto the stack as illustrated in Figure 4–12. This instruction requires 16 bytes of stack memory space to store all eight 16-bit registers. After all registers are pushed, the contents of the SP register are decremented by 16. The PUSHA instruction is very useful when the entire register set (*microprocessor environment*) of the 80286–80486 or Pentium must be saved during a task. Note that the PUSHAD instruction places the 32-bit register on the stack in the 80386/80486/Pentium.

The PUSH immediate data instruction has two different opcodes, but in both cases a 16-bit immediate number moves onto the stack, or if PUSHD is used, a 32-bit immediate datum is pushed. If the value of the immediate data is 00H–FFH, the opcode is a 6AH; if the data are 0100H–FFFFH, the opcode is 68H. The PUSH 8 instruction assembles as a 6A08H and the PUSH 1000H instruction assembles as 680010H.

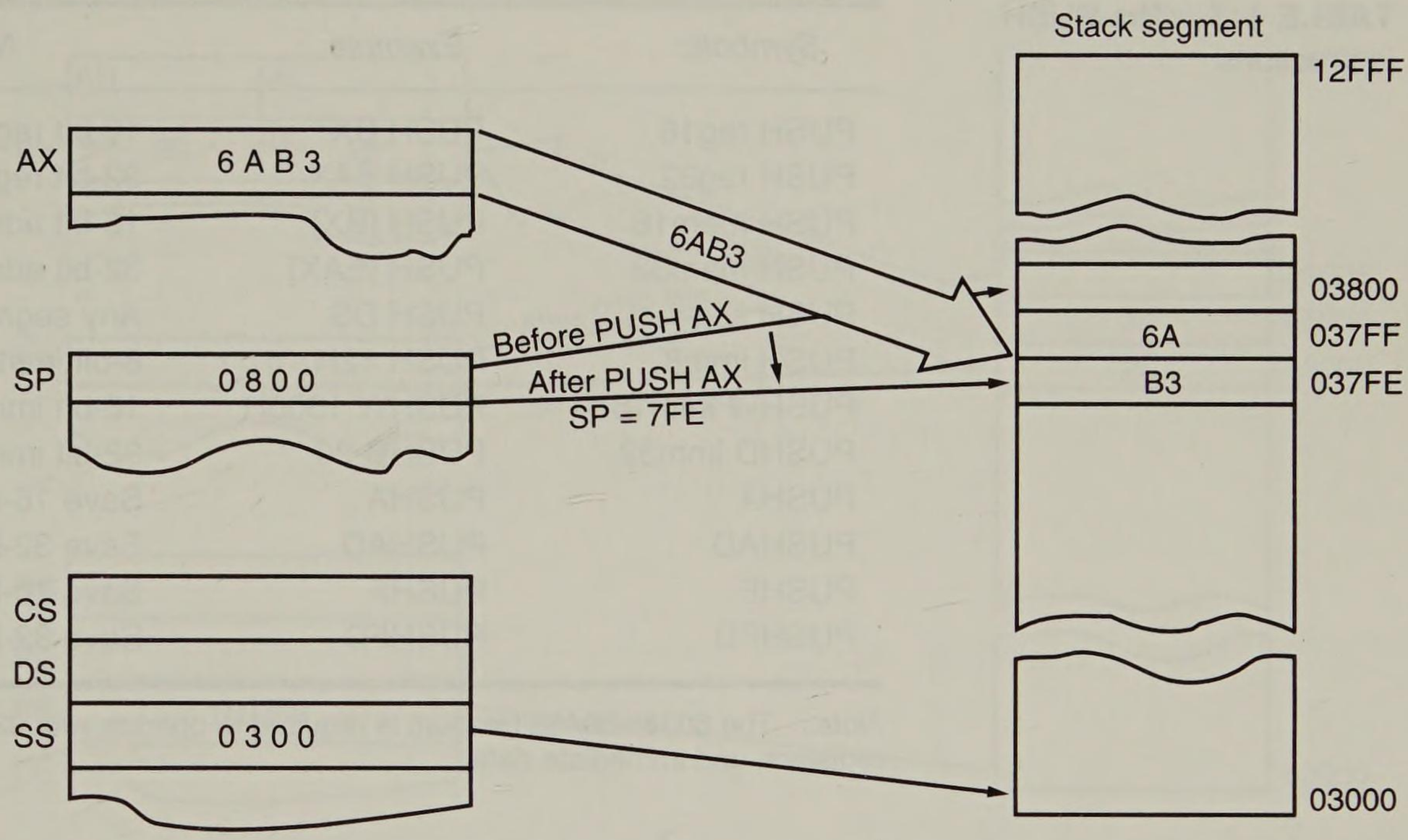



**FIGURE 4–11** The effect of a PUSH AX instruction on the SP register and stack memory locations 037FFH and 037FEH.

**FIGURE 4–12** The stack format for the PUSHA instruction.

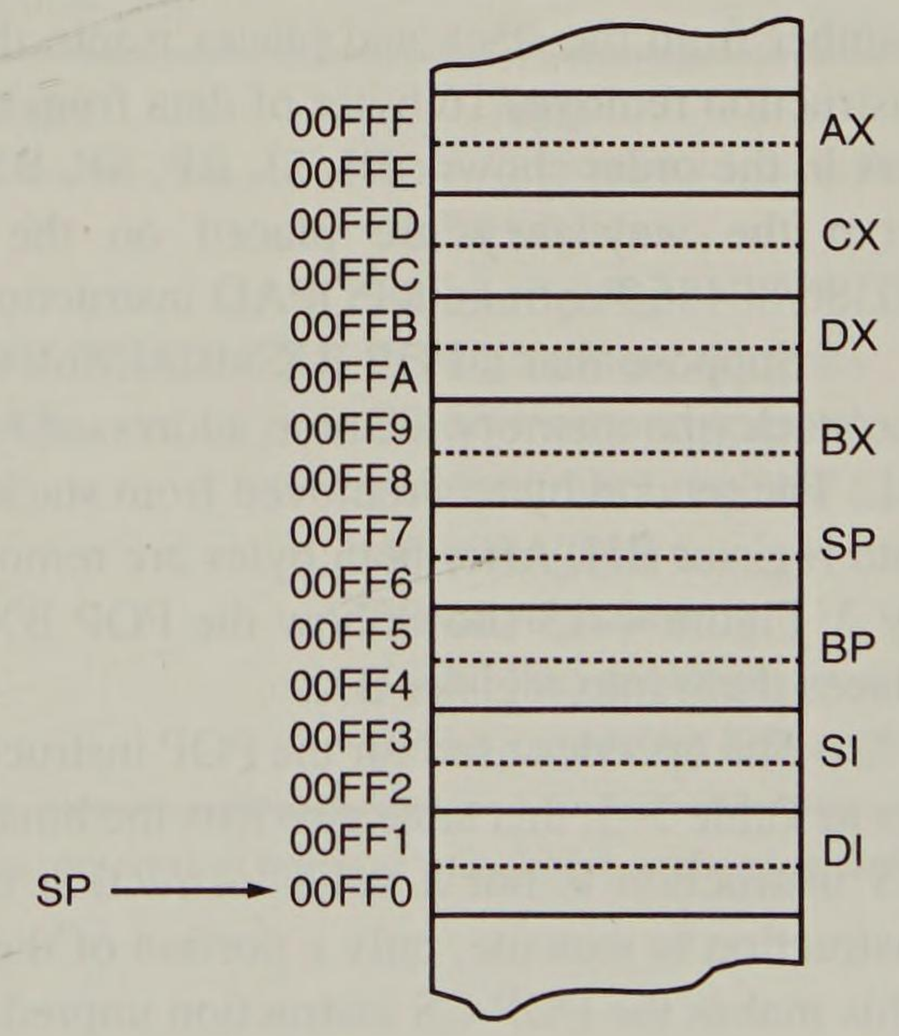



Table 4–7 lists the forms of the PUSH instruction that include PUSHA and PUSHF. Notice how the instruction set is used to specify different data sizes with the assembler.

## POP

The POP instruction performs the inverse operation of a PUSH instruction. The POP instruction removes data from the stack and places it into the target 16-bit register, seg-

**TABLE 4–7** The PUSH instructions

| *Symbolic* | *Example* | *Note* |
|---|---|---|
| PUSH reg16 | PUSH BX | 16-bit register |
| PUSH reg32 | PUSH EAX | 32-bit register |
| PUSH mem16 | PUSH [BX] | 16-bit addressing mode |
| PUSH mem32 | PUSH [EAX] | 32-bit addressing mode |
| PUSH seg | PUSH DS | Any segment register |
| PUSH imm8 | PUSH 12H | 8-bit immediate data |
| PUSHW imm16 | PUSHW 1000H | 16-bit immediate data |
| PUSHD imm32 | PUSHD 20 | 32-bit immediate data |
| PUSHA | PUSHA | Save 16-bit registers |
| PUSHAD | PUSHAD | Save 32-bit registers |
| PUSHF | PUSHF | Save 16-bit flag register |
| PUSHFD | PUSHFD | Save 32-bit flag register |

*Note*: The 80386/80486/Pentium is required to operate with 32-bit addresses, registers, and immediate data.

ment register, or a 16-bit memory location. In the 80386/80486/Pentium a POP can also remove 32-bit data from the stack and use a 32-bit address. The POP instruction is not available as an immediate POP. The POPF (*pop flags*) instruction removes a 16-bit number from the stack and places it into the flag register and the POPFD removes a 32-bit number from the stack and places it into the extended flag register. The POPA (*pop all*) instruction removes 16 bytes of data from the stack and places it into the following registers in the order shown: DI, SI, BP, SP, BX, DX, CX, and AX. This is the reverse order from the way they are placed on the stack by the PUSHA instruction. In the 80386/80486/Pentium, a POPAD instruction reloads the 32-bit registers from the stack.

Suppose that a POP BX instruction executes. The first byte of data removed from the stack (the memory location addressed by SP in the stack segment) moves into register BL. The second byte is removed from stack segment memory location SP + 1, and placed into register BH. After both bytes are removed from the stack, the SP register increments by 2. Figure 4–13 shows how the POP BX instruction removes data from the stack and places them into register BX.

The opcodes used for the POP instruction, and all its variations, appear in Table 4–8. As in Table 3–5, this table also lists the binary machine language opcodes. Note that a POP CS instruction is not a valid instruction in the instruction set. If we allow a POP CS instruction to execute, only a portion of the address (CS) of the next instruction changes. This makes the POP CS instruction unpredictable and therefore not allowed.

## Initializing the Stack

When the stack area is initialized, we load both the stack segment register (SS) and the stack pointer register (SP). It is normal to designate an area of memory as the stack segment by loading SS with the bottom location of the stack segment.

For example, if the stack segment is to reside in memory locations 10000H–1FFFFH, we load SS with a 1000H. (Recall that we append the right-most end

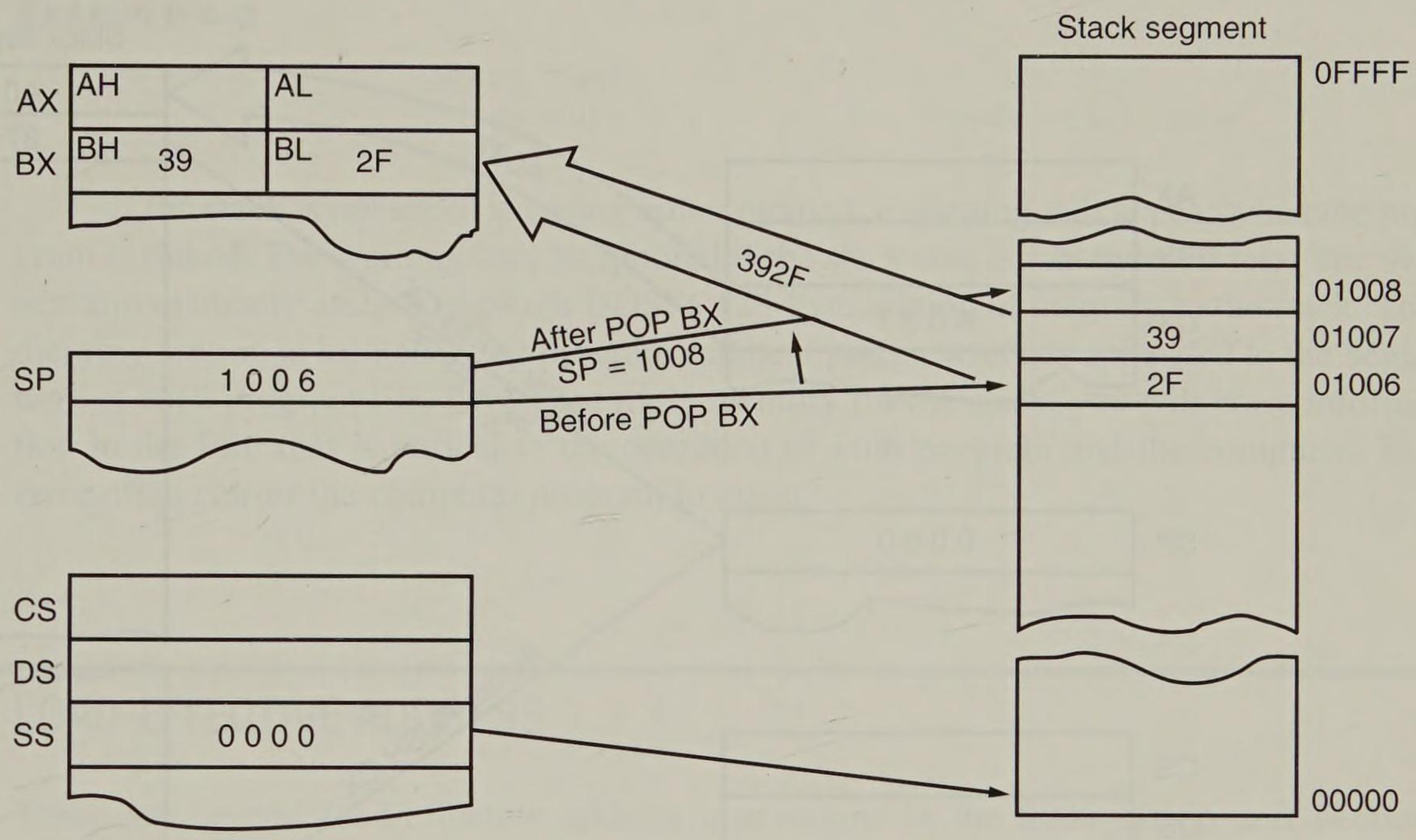



**FIGURE 4–13** POP BX removes two bytes of data from the stack segment and places them into the BX register.

**TABLE 4–8** The POP instructions

| *Symbolic* | *Example* | *Note* |
|---|---|---|
| POP reg16 | POP DI | 16-bit register |
| POP reg32 | POP EBX | 32-bit register |
| POP mem16 | POP WORD PTR[DI+2] | 16-bit memory address |
| POP mem32 | POP DATA3 | 32-bit memory address |
| POP seg | POP GS | Any segment register |
| POPA | POPA | 16-bit registers |
| POPAD | POPAD | 32-bit registers |
| POPF | POPF | 16-bit flag register |
| POPFD | POPFD | 32-bit extended flag register |

*Note*: The 80386/80486/Pentium is required to operate with 32-bit addresses and registers.

of the stack segment register with a 0H for real-mode addressing.) To start the stack at the top of this 64K byte stack segment, the stack pointer (SP) is loaded with a 0000H. Likewise, to address the top of the stack at location 10FFFH we use a value of 1000H in SP. Figure 4–14 shows how this value causes data to be pushed onto the top of the stack segment with a PUSH CX instruction. Remember that all segments are *cyclic* in nature; that is, the top location of a segment is *contiguous* with the bottom location of the segment.

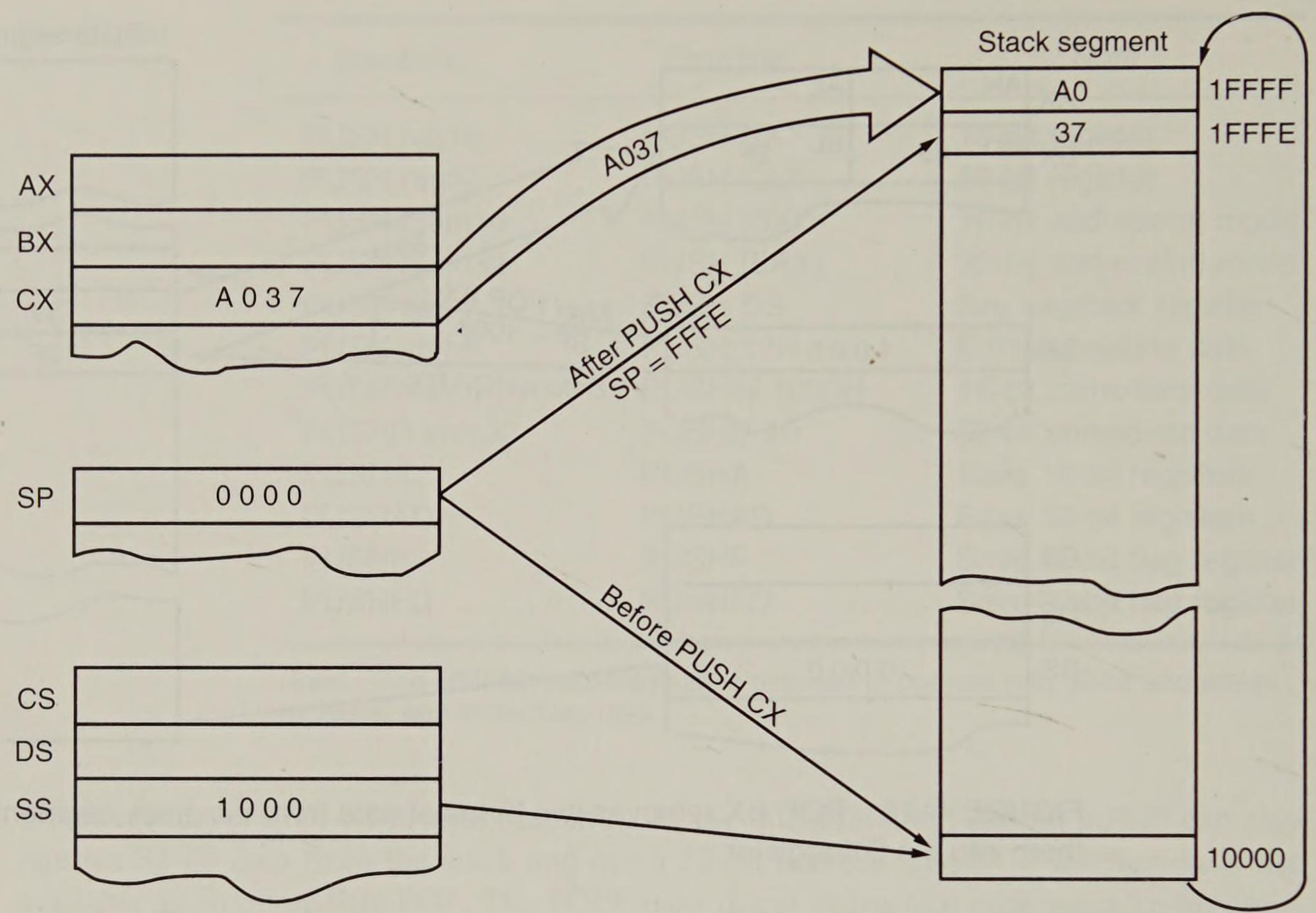



**FIGURE 4–14** PUSH CX illustrating the cyclic nature of the stack segment. Notice that SP starts at 0000H and, after being decremented, ends up at FFFEH.

**EXAMPLE 4–1**

```
0000                    STACK_SEG   SEGMENT STACK

0000 0100[                          DW   100H DUP (?)
           ????
                ]
0200                    STACK_SEG   ENDS
```

In assembly language, a stack segment is set up as illustrated in Example 4–1. The first statement identifies the start of the stack segment and the last statement identifies the end of the stack segment. The assembler and linker program places the correct stack segment address in SS and the length of the segment (top of the stack) into SP. There is no need to load these registers in your program unless you wish to change the initial values for some reason.

An alternative method for defining the stack segment is used with one of the memory models for the MASM assembler only (refer to Appendix A). Other assemblers do not use models, or if they do they are not exactly the same as with MASM. Here (see Example 4–2) the .STACK statement, followed by the number of *bytes* allocated to the stack, defines the stack area. The function is identical to Example 4–1. The .STACK statement also initializes both SS and SP.

**EXAMPLE 4–2**

```
.MODEL SMALL
.STACK 200H                          ;set stack size
```

If the stack is not specified using either method, a warning will appear when the program is linked. The warning may be ignored if the stack size is 128 bytes or less. The system automatically assigns (through DOS) a 128-byte section of memory to the stack. This memory section is located in the program segment prefix, which is appended to the beginning of each program file. If you use more memory for the stack, you will erase information in the PSP that is critical to the operation of your program and the computer. This error often causes the computer program to crash.

## 4–3 LOAD-EFFECTIVE ADDRESS

There are several load-effective address instructions in the 8086–80486 and Pentium microprocessor instruction set. The LEA instruction loads any 16-bit register with the address as determined by the addressing mode selected for the instruction. The LDS and LES variations load any 16-bit register with the offset address retrieved from a memory location and then load either DS or ES with a segment address retrieved from memory. In the 80386, 80486, and Pentium microprocessors LFS, LGS, and LSS are added to the instruction set and a 32-bit register can be selected to receive a 32-bit offset from memory. Table 4–9 lists the load-effective address instructions.

### LEA

The LEA instruction loads a 16-bit register with the offset address of the data specified by the operand. As the first example in Table 4–9 shows, the operand address NUMB, not the contents of address NUMB, is loaded into register AX.

**TABLE 4–9** The load-effective address instructions

| *Symbolic* | *Function* |
|---|---|
| LEA AX,NUMB | AX is loaded with the address of NUMB |
| LEA EAX,NUMB | EAX is loaded with the address of NUMB |
| LDS DI,LIST | DI and DS are loaded with the address stored at LIST |
| LDS EDI,LIST | EDI and DS are loaded with the address stored at LIST |
| LES BX,CAT | BX and ES are loaded with the address stored at CAT |
| LFS DI,DATA1 | DI and FS are loaded with the address stored at DATA1 |
| LGS SI,DATA5 | SI and GS are loaded with the address stored at DATA5 |
| LSS SP,MEM | SP and SS are loaded with the address stored at MEM |

By comparing LEA with MOV, we observe the following effect: LEA BX,[DI] loads the offset address specified by [DI] (contents of DI) into the BX register; MOV BX,[DI] loads the data stored at the memory location addressed by [DI] into register BX.

Earlier in the text, we presented several examples using the OFFSET pseudo-operation. The *OFFSET directive* performs the same function as an LEA instruction if the operand is a displacement. For example, MOV BX,OFFSET LIST performs the same function as LEA BX,LIST. Both instructions load the offset address of memory location LIST into the BX register.

But why is the LEA instruction available if the OFFSET directive accomplishes the same task? First, OFFSET only functions with simple operands such as LIST. It may not be used for an operand such as [DI], LIST [SI], etc. The OFFSET directive is more efficient than the LEA instruction for simple operands. It takes the microprocessor longer to execute LEA BX,LIST than MOV BX,OFFSET LIST. The 80286, for example, requires three clocks to execute the LEA BX,LIST instruction and only two clocks to execute MOV BX,OFFSET LIST. The reason that the MOV BX,OFFSET LIST instruction executes faster is because the assembler calculates the offset address of LIST, while with the LEA instruction, the microprocessor does the calculation as it executes the instruction. The MOV BX,OFFSET LIST instruction is actually assembled as a move immediate instruction and is more efficient.

Suppose that the microprocessor executes an LEA BX,[DI] instruction and DI contains a 1000H. Because DI contains the offset address, the microprocessor transfers a copy of DI into BX. A MOV BX,DI instruction performs this task in less time and is often preferred to the LEA BX,[DI] instruction.

Another example is LEA SI,[BX + DI]. This instruction adds BX to DI and stores the sum in the SI register. The sum generated is a modulo-64K sum. If BX = 1000H and DI = 2000H, the offset address moved into SI is 3000H. If BX = 1000H and DI = FF00H, the offset address is 0F00H instead of 10F00H. Notice that the second result is a modulo-64K sum of 0F00H. (A modulo-64K sum drops out any carry from the 16-bit result.)

## LDS, LES, LFS, LGS, and LSS

The LDS, LES, LFS, LGS, and LSS instructions load any 16-bit or 32-bit register with an offset address and the DS, ES, FS, GS, or SS segment register with a segment address. These instructions use any of the memory addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. These instructions may not use the register addressing mode (MOD = 11). Note that the LFS, LGS, and LSS instructions are only available on 80386, 80486, and Pentium microprocessors, as are the 32-bit registers.

Figure 4–15 illustrates an example LDS BX,[DI] instruction. This instruction transfers the 32-bit number, addressed by DI in the data segment, into the BX and DS registers. The LDS, LES, LFS, LGS, and LSS instructions obtain a new far address from memory. The offset address appears first, followed by the segment address. This format is used for storing all 32-bit memory addresses.

A far address can be stored in memory by the assembler. For example, the ADDR DD FAR PTR FROG instruction stores the offset and segment address (*far address*) of FROG in 32 bits of memory at location ADDR. The DD directive tells the assembler to store a *doubleword* (32-bit number) in memory address ADDR.

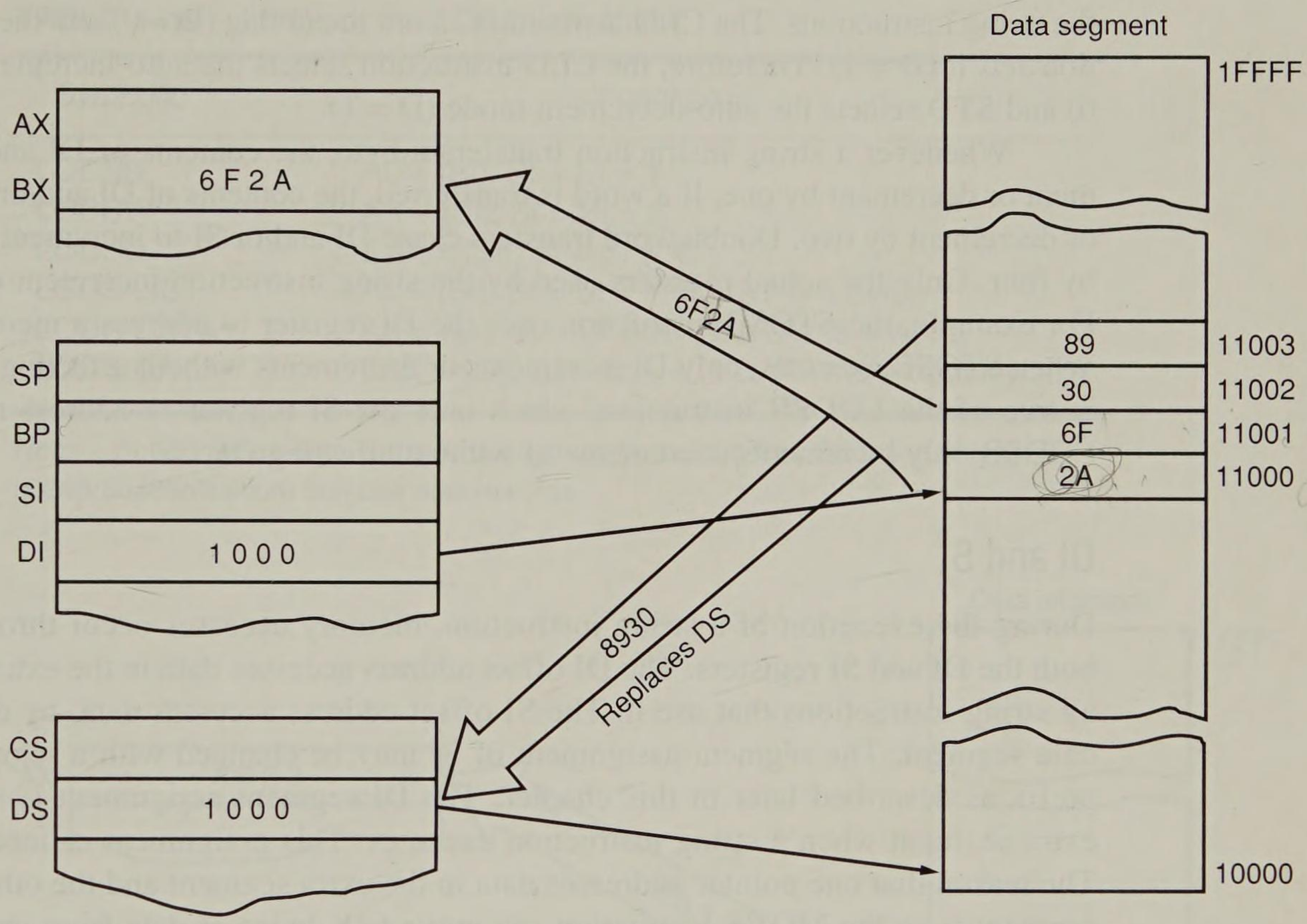



**FIGURE 4–15** LDS BX,[DI] loads the BX register from locations 11000H and 11001H and the DS register from locations 11002H and 11003H. This instruction will change the data segment from 10000H–1FFFFH to 89300H–992FFH.

In the 80386/80486/Pentium, an LDS EBX,[DI] instruction loads EBX from the 4-byte section of memory addressed by DI in the data segment. Following this 4-byte offset is a word that is loaded to the DS register. Notice that instead of addressing a 32-bit section of memory, the 80386/80486/Pentium addresses a 48-bit section of the memory whenever a 32-bit offset address is loaded to a 32-bit register.

## 4–4 STRING DATA TRANSFERS

There are five string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS. Each string instruction allows data transfers that are either a single byte, word, or doubleword or, if repeated, a block of bytes, words, or doublewords. Before the string instructions are presented, the operation of the D flag bit (*direction*), DI, and SI must be understood as they apply to the string instructions.

### The Direction Flag

The direction flag (D) selects auto-increment (D = 0) or auto-decrement (D = 1) operation for the DI and SI registers during string operations. The direction flag is only used with

the string instructions. The CLD instruction *clears* the D flag (D = 0) and the STD instruction *sets* it (D = 1). Therefore, the CLD instruction selects the auto-increment mode (D = 0) and STD selects the auto-decrement mode (D = 1).

Whenever a string instruction transfers a byte, the contents of DI and/or SI increment or decrement by one. If a word is transferred, the contents of DI and/or SI increment or decrement by two. Doubleword transfers cause DI and/or SI to increment or decrement by four. Only the actual registers used by the string instruction increment or decrement. For example, the STOSB instruction uses the DI register to address a memory location. When STOSB executes, only DI increments or decrements without affecting SI. The same is true of the LODSB instruction, which uses the SI register to address memory data. LODSB only increments/decrements SI without affecting DI.

## DI and SI

During the execution of a string instruction, memory accesses occur through either or both the DI and SI registers. The DI offset address accesses data in the extra segment for all string instructions that use it. The SI offset address accesses data, by default, in the data segment. The segment assignment of SI may be changed with a segment override prefix as described later in this chapter. The DI segment assignment is *always* in the extra segment when a string instruction executes. This assignment cannot be changed. The reason that one pointer addresses data in the extra segment and the other in the data segment is so the MOVS instruction can move 64K bytes of data from one segment of memory to another.

## LODS

The LODS instruction loads AL, AX, or EAX with data stored at the data segment offset address indexed by the SI register. (Note that only the 80386, 80486, and Pentium can use EAX.) After loading AL with a byte, AX with a word, or EAX with a doubleword, the contents of SI increment if D = 0, or decrement if D = 1. A 1 is added to or subtracted from SI for a byte-sized LODS, a 2 is added to or subtracted from SI for a word-sized LODS, and a 4 is added to or subtracted from SI for a doubleword-sized LODS.

Table 4–10 lists the permissible forms of the LODS instruction. The LODSB (*loads a byte*) instruction causes a byte to be loaded into AL, the LODSW (*loads a word*) instruction causes a word to be loaded into AX, and the LODSD (*loads a doubleword*) instruction causes a doubleword to be loaded into EAX. Alternatively, the LODS instruction may be followed by a byte-, word-, or doubleword-sized operand to select a byte, word, or doubleword transfer, but this is unusual. Operands are often defined as bytes with DB, as words with DW, and as doublewords with DD. The DB pseudo-operation defines byte(s), the DW pseudo-operation defines word(s), and the DD pseudo-operation defines doubleword(s).

Figure 4–16 shows the effect of executing the LODSW instruction if the D flag = 0, SI = 1000H, and DS = 1000H. Here a 16-bit number, stored at memory locations 11000H and 11001H, moves into AX. Because D = 0, and this is a word transfer, the contents of SI increment by 2 *after* AX loads with memory data.

**TABLE 4–10** Forms of the LODS instruction

| *Symbolic* | *Function* |
|---|---|
| LODSB | AL = [SI]; SI = SI ± 1 |
| LODSW | AX = [SI]; SI = SI ± 2 |
| LODSD | EAX = [SI]; SI = SI ± 4 |
| LODS LIST | AL = [SI]; SI = SI ± 1 (if LIST is a byte) |
| LODS DATA1 | AX = [SI]; SI = SI ± 2 (if DATA1 is a word) |
| LODS DATA4 | EAX = [SI]; SI = SI ± 4 (if DATA4 is a doubleword) |

*Note*: SI addresses data in the data segment by default for LODS and only the 80386/80486/Pentium can use doublewords.

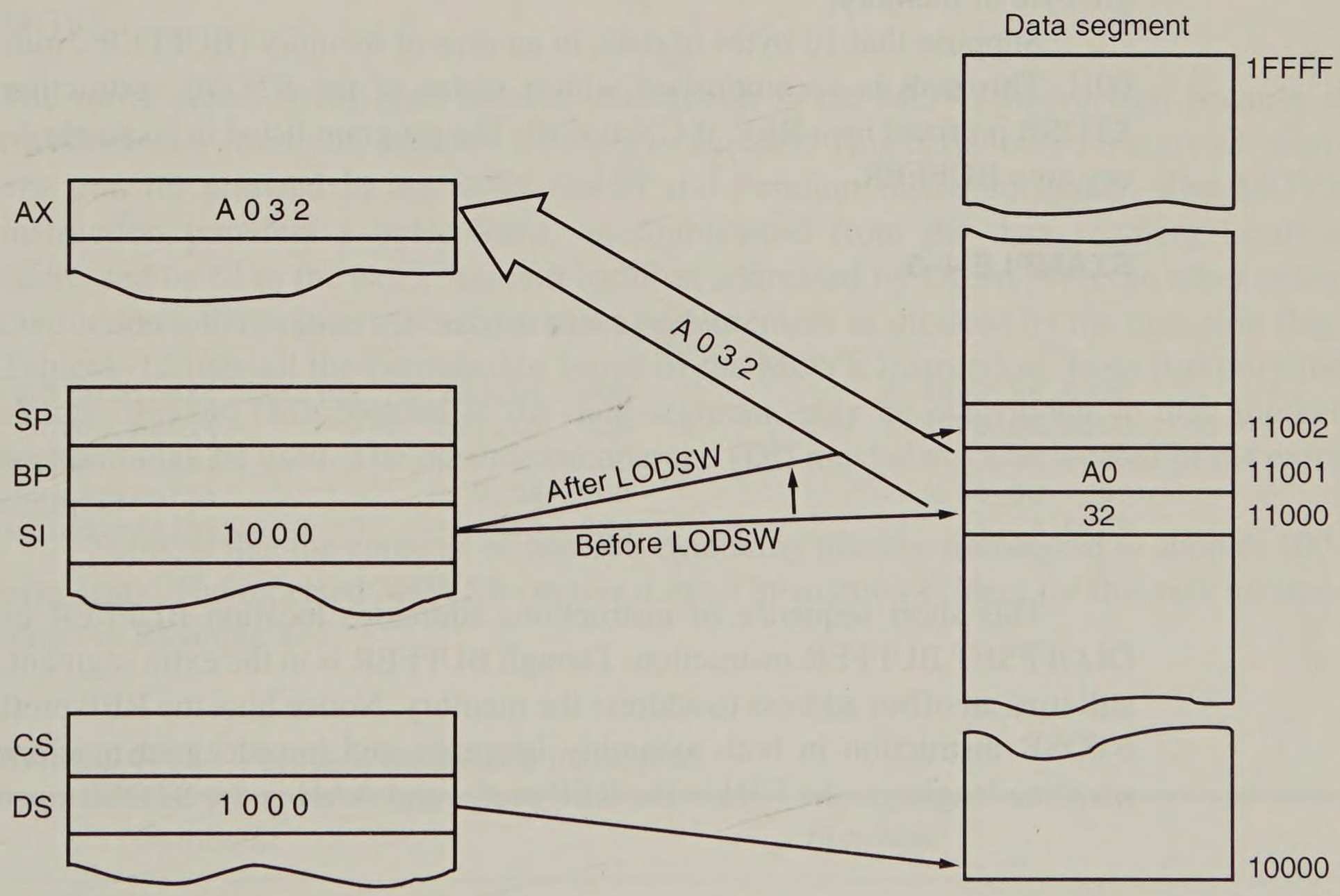



**FIGURE 4–16** The effect of executing the LODSW instruction if DS = 1000H, SI = 1000H, D = 0, 11000H = 32, and 11001H = A0.

## STOS

The STOS instruction stores AL, AX, or EAX at the extra segment memory location addressed by the DI register. (Note that only the 80386/80486/Pentium can use EAX and doublewords.) Table 4–11 lists all forms of the STOS instruction. As with LODS, a STOS instruction may be appended with a B, W, or D for a byte, word, or doubleword transfer. The STOSB (*stores a byte*) instruction stores the byte in AL at the extra segment memory location addressed by DI. The STOSW (*stores a word*) instruction stores AX in the

extra segment memory location addressed by DI. A doubleword is stored in the extra segment location addressed by DI with the STOSD (*stores a doubleword*) instruction. After the byte (AL), word (AX), or doubleword (EAX) is stored, the contents of DI increment or decrement.

*STOS with a REP.* The *repeat prefix* (REP) may be added to any string data transfer instruction except the LODS instruction. The REP prefix causes CX to decrement by 1 each time the string instruction executes. After CX decrements, the string instruction repeats. If CX reaches a value of 0, the instruction terminates and the program continues with the next sequential instruction. Thus, if CX is loaded with a 100, and a REP STOSB instruction executes, the microprocessor automatically repeats the STOSB instruction 100 times. Since the DI register is automatically incremented or decremented after each datum is stored, this instruction stores the contents of AL in a block of memory instead of a single byte of memory.

Suppose that 10 bytes of data, in an area of memory (BUFFER), must be cleared to 00H. This task is accomplished with a series of ten STOSB instructions or with one STOSB prefixed by a REP, if CX is a 10. The program listed in Example 4–3 clears memory area BUFFER.

**EXAMPLE 4–3**

```
                        ;Using a REP STOSB to clear a BUFFER
                        ;
0000 BF 0000 R          MOV  DI,OFFSET BUFFER  ;address BUFFER
0003 B9 000A            MOV  CX,10             ;load count
0006 FC                 CLD                    ;auto-increment
0007 B0 00              MOV  AL,0              ;clear AL
0009 F3/AA              REP  STOSB             ;clear buffer
```

This short sequence of instructions addresses location BUFFER using the MOV DI,OFFSET BUFFER instruction. Though BUFFER is in the extra segment, the assembler still uses an offset address to address the memory. Notice how the REP prefix precedes the STOSB instruction in both assembly language and hexadecimal machine language. In machine language, the F3H is the REP prefix and AAH is the STOSB opcode.

**TABLE 4–11** Forms of the STOS instruction

| *Symbolic* | *Function* |
|---|---|
| STOSB | [DI] = AL; DI = DI ± 1 |
| STOSW | [DI] = AX; DI = DI ± 2 |
| STOSD | [DI] = EAX; DI = DI ± 4 |
| STOS LIST | [DI] = AL; DI = DI ± 1 (if LIST is a byte) |
| STOS DATA1 | [DI] = AX; DI = DI ± 2 (if DATA1 is a word) |
| STOS DATA4 | [DI] = EAX; DI = DI ± 4 (if DATA4 is a doubleword) |

*Note*: DI addresses data in the extra segment, and doublewords are only used by the 80386, 80486, or Pentium microprocessor.

A faster method of clearing this 10-byte buffer is to use the STOSW instruction with a count of 5. Example 4–4 illustrates the same task as Example 4–3, except the count changes to a 5 and the STOSW instruction repeats instead of the STOSB instruction. Also, register AX is cleared instead of register AL.

**EXAMPLE 4–4**

```
                        ;Using a REP STOSW to clear a BUFFER
                        ;
0000 BF 0000 R                  MOV  DI,OFFSET BUFFER  ;address BUFFER
0003 B9 0005                    MOV  CX,5              ;load count
0006 FC                         CLD                    ;auto-increment
0007 B8 0000                    MOV  AX,0              ;clear AX
000A F3/AB                      REP  STOSW             ;clear buffer
```

## MOVS

The most useful string data transfer instruction is the MOVS instruction because it transfers data from one memory location to another. This is the only *memory-to-memory* transfer allowed in the 8086–80486 and Pentium microprocessors. The MOVS instruction transfers a byte, word, or doubleword from the data segment location addressed by SI to the extra segment location addressed by DI. As with the other string instructions, the pointers then increment or decrement as dictated by the direction flag. Table 4–12 lists all the permissible forms of the MOVS instruction. Note that only the source operand (SI), located in the data segment, may be overridden so that another segment may be used. The destination operand (DI) must always be located in the extra segment.

Suppose that the contents of one 100-byte array must be transferred to another 100-byte array. The repeated MOVSB (*moves a byte*) instruction is ideal for this task as illustrated in Example 4–5.

**TABLE 4–12** Forms of the MOVS instruction

| *Symbolic* | *Function* |
|---|---|
| MOVSB | [DI] = [SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred) |
| MOVSW | [DI] = [SI]; DI = DI ± 2; SI = SI ± 2 (word transferred) |
| MOVSD | [DI] = [SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred) |
| MOVS BYTE1,BYTE2 | [DI] = [SI]; DI = DI ± 1; SI = SI ± 1 (if BYTE1 and BYTE2 are bytes) |
| MOVS WORD1,WORD2 | [DI] = [SI]; DI = DI ± 2; SI = SI ± 2 (if WORD1 and WORD2 are words) |
| MOVS DWORD1,DWORD2 | [DI] = [SI]; DI = DI ± 4; SI = SI ± 4 (if DWORD1 and DWORD2 are doublewords) |

**EXAMPLE 4–5**

```
                        ;Using a REP MOVSB to transfer data from LIST1 into LIST2
                        ;
0000 BE 0000 R                MOV  SI,OFFSET LIST1   ;address LIST1
0003 BF 0064 R                MOV  DI,OFFSET LIST2   ;address LIST2
0006 B9 0064                  MOV  CX,100            ;load count
0009 FC                       CLD                    ;auto-increment
000A F3/A4                    REP  MOVSB             ;transfer data
```

## INS

The INS (*input string*) instruction (not available on the 8086/8088 microprocessor) transfers a byte, word, or doubleword of data from an I/O device into the extra segment memory location addressed by the DI register. The *I/O address* is contained in the DX register. This instruction is useful for inputting a block of data from an external I/O device directly into the memory. One application transfers data from a disk drive to memory. Disk drives are often considered and interfaced as I/O devices in a computer system.

As with the prior string instructions, there are two basic forms of the INS. The INSB instruction inputs data from an 8-bit I/O device and stores it in the byte-sized memory location indexed by SI. The INSW instruction inputs 16-bit I/O data and stores it in a word-sized memory location. The INSD instruction inputs a doubleword. These instructions can be repeated using the REP prefix. This allows an entire block of input data to be stored in the memory from an I/O device. Table 4–13 lists the various forms of the INS instruction.

Example 4–6 shows a short program that inputs 50 bytes from an I/O device whose address is 03ACH and stores the data in memory array LISTS. This software assumes that data are available from the I/O device at all times. Otherwise, the software must check to see if the I/O device is ready to transfer data, precluding the use of a REP prefix.

**EXAMPLE 4–6**

```
                        ;Using the REP INSB to input data to a memory array
                        ;
0000 BF 0000 R                MOV  DI,OFFSET LISTS   ;address array
0003 BA 03AC                  MOV  DX,3ACH           ;address I/O
0006 FC                       CLD                    ;auto-increment
0007 B9 0032                  MOV  CX,50             ;load count
000A F3/6C                    REP  INSB              ;input data
```

## OUTS

The OUTS (*output string*) instruction (not available on the 8086/8088 microprocessor) transfers a byte, word, or doubleword of data from the data segment memory location address by SI to an I/O device. The I/O device is addressed by the DX register as it was with the INS instruction. Table 4–14 shows the variations available for the OUTS instruction.

Example 4–7 shows a short program that transfers data from a memory array to an I/O device. This software assumes that the I/O device is always ready for data.

**TABLE 4–13** Forms of the INS instruction

| *Symbolic* | *Function* |
|---|---|
| INSB | [DI] = [DX]; DI = DI ± 1 (byte transferred) |
| INSW | [DI] = [DX]; DI = DI ± 2 (word transferred) |
| INSD | [DI] = [DX]; DI = DI ± 4 (doubleword transferred) |
| INS LIST | [DI] = [DX]; DI = DI ± 1 (if LIST is a byte) |
| INS DATA1 | [DI] = [DX]; DI = DI ± 2 (if DATA1 is a word) |
| INS DATA4 | [DI] = [DX]; DI = DI ± 4 (if DATA4 is a doubleword) |

*Note*: [DX] indicates that DX contains the I/O device address. These instructions are not available on the 8086/8088 microprocessor. Only the 80386/80486/Pentium uses doublewords.

**TABLE 4–14** Forms of the OUTS instruction

| *Symbolic* | *Function* |
|---|---|
| OUTSB | [DX] = [SI]; SI = SI ± 1 (byte transferred) |
| OUTSW | [DX] = [SI]; SI = SI ± 2 (word transferred) |
| OUTSD | [DX] = [SI]; SI = SI ± 4 (doubleword transferred) |
| OUTS LIST | [DX] = [SI]; SI = SI ± 1 (if LIST is a byte) |
| OUTS DATA1 | [DX] = [SI]; SI = SI ± 2 (if DATA1 is a word) |
| OUTS DATA4 | [DX] = [SI]; SI = SI ± 4 (if DATA4 is a doubleword) |

*Note*: [DX] indicates that DX contains the I/O device address. These instructions are not available on the 8086/8088 microprocessor. Only the 80386/80486/Pentium uses doublewords.

**EXAMPLE 4–7**

```
                        ;Using the REP OUTS to output data from a memory array
                        ;
0000 BE 0064 R                MOV  SI,OFFSET ARRAY   ;address array
0003 BA 03AC                  MOV  DX,3ACH           ;address I/O
0006 FC                       CLD                    ;auto-increment
0007 B9 0064                  MOV  CX,100            ;load count
000A F3/6E                    REP  OUTSB
```

## 4–5 MISCELLANEOUS DATA TRANSFER INSTRUCTIONS

Don't be fooled by the term miscellaneous; these instructions are used in programs. The data transfer instructions detailed in this section are: XCHG, LAHF, SAHF, XLAT, IN, OUT, BSWAP, MOVSX, and MOVZX. Because the miscellaneous instructions are not be used as often as a MOV instruction, they have been grouped together in this section.

## XCHG

The exchange instruction (XCHG) exchanges the contents of a register with the contents of any other register or memory location. The XCHG instruction cannot exchange segment registers or memory-to-memory data. Exchanges are byte-, word-, or doubleword-sized (80386/80486/Pentium only), and use any addressing mode discussed in Chapter 3 except immediate addressing. Table 4–15 shows the forms available for the XCHG instruction.

The XCHG instruction, using the 16-bit AX register with another 16-bit register, is the most efficient exchange. This instruction occupies one byte of memory. Other XCHG instructions require two or more bytes of memory depending on the addressing mode selected.

When using a memory addressing mode and the assembler, it doesn't matter which operand addresses memory. The XCHG AL,[DI] instruction is identical to the XCHG [DI],AL instruction as far as the assembler is concerned.

If the 80386/80486/Pentium microprocessor is available, the XCHG instruction can exchange doubleword data. For example, the XCHG EAX,EBX instruction exchanges the contents of the EAX register with the EBX register.

## LAHF and SAHF

The LAHF and SAHF instructions are seldom used because they were designed as *bridge* instructions. These instructions allowed 8085 (an early 8-bit microprocessor) software to be translated into 8086 software by a translation program. Because any translation required by a particular software was probably completed years ago, these instructions have little application today. The LAHF instruction transfers the right-most eight bits of the flag register into the AH register. The SAHF instruction transfers the AH register into the right-most eight bits of the flag register.

At times the SAHF instruction may find some application with the numeric coprocessor. The numeric coprocessor contains a status register than can be copied from AH into the flag register. This can be used to test some of the conditions of the numeric coprocessor. This is detailed in the chapter that explains the operation and programming of the numeric coprocessor.

## XLAT

The XLAT (*translate*) instruction converts the contents of the AL register into a number stored in a memory table. This instruction performs the direct table lookup technique often

**TABLE 4–15** Forms of the XCHG instruction

| *Symbolic* | *Note* |
|---|---|
| XCHG reg,reg | Exchanges byte, word, or doubleword registers |
| XCHG reg,mem | Exchanges byte, word, or doubleword memory data with register data |

*Note*: Only the 80386/80486/Pentium uses doubleword data.

used to convert one code to another. An XLAT instruction first adds the contents of AL to BX to form a memory address within the data segment. It then copies the contents of this address into AL. This is the only instruction that adds an 8-bit number to a 16-bit number.

Suppose that a seven-segment LED display lookup table is stored in memory at address TABLE. The XLAT instruction translates the BCD number in AL to a seven-segment code in AL. Example 4–8 provides a short program that converts from a BCD code to seven-segment code. Figure 4–17 shows the operation of this example program if TABLE = 1000H, DS = 1000H, and the initial value of AL = 05H (a 5 BCD). After the translation, AL = 6DH.

**EXAMPLE 4–8**

```
                        ;Using an XLAT to convert from BCD to 7-segment code
                        ;
0000 BB 1000 R                MOV  BX,OFFSET TABLE   ;address TABLE
0003 D7                       XLAT                   ;convert
```

## IN and OUT

Table 4–16 lists the forms of the IN and OUT instructions, which perform I/O operations. Notice that the contents of AL, AX, or EAX *only* are transferred between the I/O device and the microprocessor. An IN instruction transfers data from an external I/O device to

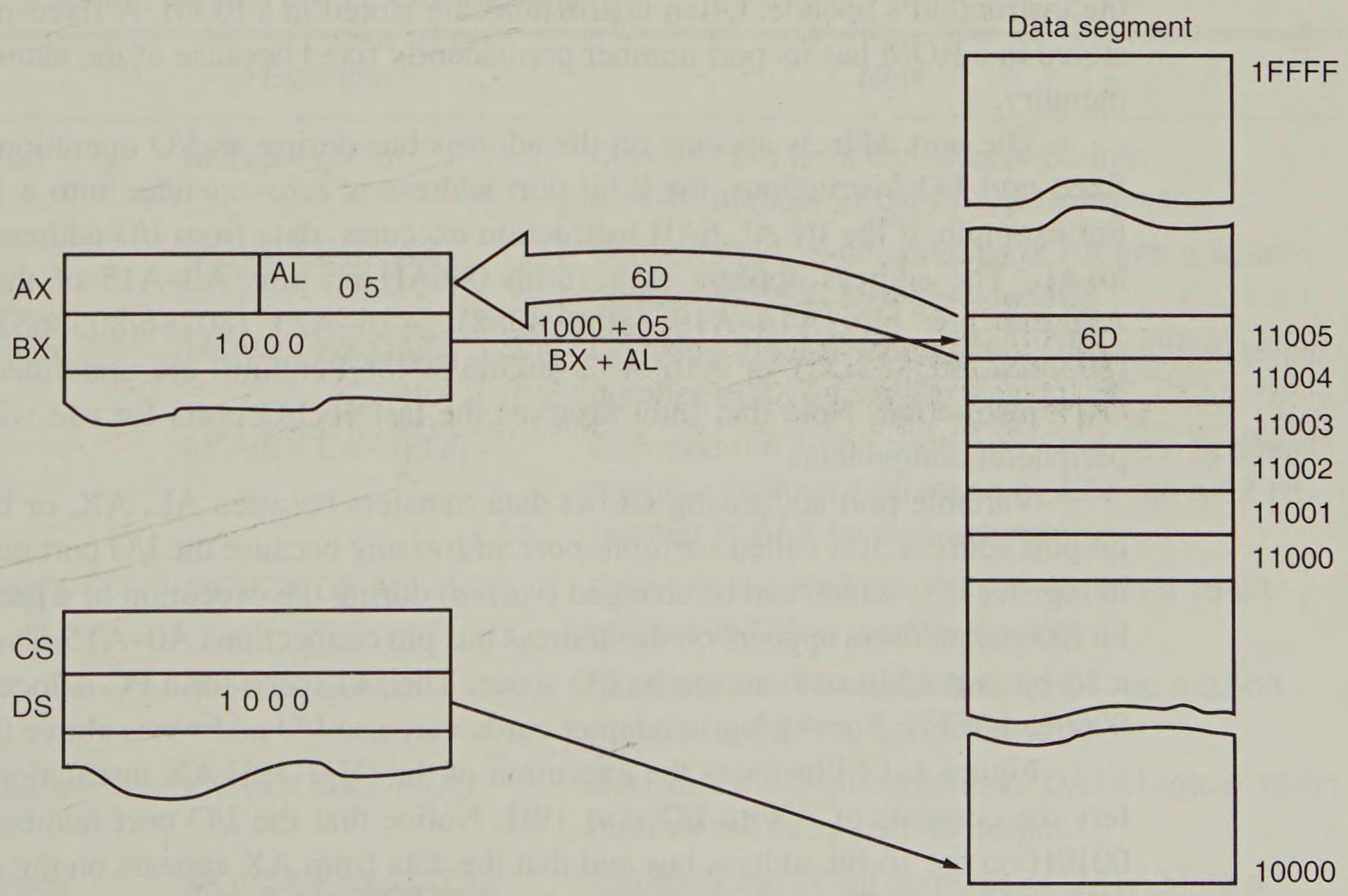



**FIGURE 4–17** The effect of executing the XLAT instruction at the point just before the 6DH from memory location 11005H is gated into the AL register.

**TABLE 4–16** IN and OUT instructions

| *Symbolic* | *Function* |
|---|---|
| IN AL,p8 | 8-bit data are input to AL from port p8 |
| IN AX,p8 | 16-bit data are input to AX from port p8 |
| IN EAX,p8 | 32-bit data are input to EAX from port p8 |
| IN AL,DX | 8-bit data are input to AL from port DX |
| IN AX,DX | 16-bit data are input to AX from port DX |
| IN EAX,DX | 32-bit data are input to EAX from port DX |
| OUT p8,AL | 8-bit data are sent to port p8 from AL |
| OUT p8,AX | 16-bit data are sent to port p8 from AX |
| OUT p8,EAX | 32-bit data are sent to port p8 from EAX |
| OUT DX,AL | 8-bit data are sent to port DX from AL |
| OUT DX,AX | 16-bit data are sent to port DX from AX |
| OUT DX,EAX | 32-bit data are sent to port DX from EAX |

*Note*: p8 = an 8-bit I/O port number. DX = the 16-bit port address held in DX.

AL, AX, or EAX, and an OUT instruction transfers data from AL, AX, or EAX to an external I/O device. (Note only the 80386/80486/Pentium contains EAX.)

Two forms of I/O device (*port*) address exist for IN and OUT: fixed-port and variable-port. Fixed-port addressing allows data transfer between AL, AX, or EAX using an 8-bit I/O port address. It is called *fixed-port addressing* because the port number follows the instruction's opcode. Often instructions are stored in a ROM. A fixed-port instruction stored in a ROM has its port number permanently fixed because of the nature of read-only memory.

The port address appears on the address bus during an I/O operation. For the 8-bit fixed-port I/O instructions, the 8-bit port address is zero-extended into a 16-bit address. For example, if the IN AL,6AH instruction executes, data from I/O address 6AH is input to AL. The address appears as a 16-bit 006AH on pins A0–A15 of the address bus. Address bus bits A16–A19 (8086/8088), A16–A23 (80286/80386SX), A16–A24 (80386SL/80386SLC), or A16–A32 (80386/80486/Pentium) are undefined for an IN or OUT instruction. Note that Intel reserves the last 16 I/O ports for use with some of its peripheral components.

Variable-port addressing allows data transfers between AL, AX, or EAX and a 16-bit port address. It is called *variable-port addressing* because the I/O port number is stored in register DX, which can be changed (*varied*) during the execution of a program. The 16-bit I/O port address appears on the address bus pin connections A0–A15. The IBM PC uses a 16-bit port address to access its I/O space. The I/O space for a PC is located at I/O port 0000H–03FFH. Some plug-in adapter cards may use I/O addresses above 03FFH.

Figure 4–18 illustrates the execution of the OUT 19H,AX instruction, which transfers the contents of AX to I/O port 19H. Notice that the I/O port number appears as a 0019H on the 16-bit address bus and that the data from AX appears on the data bus of the microprocessor. The system control signal $\overline{\text{IOWC}}$ (I/O write control) is a logic zero to enable the I/O device.

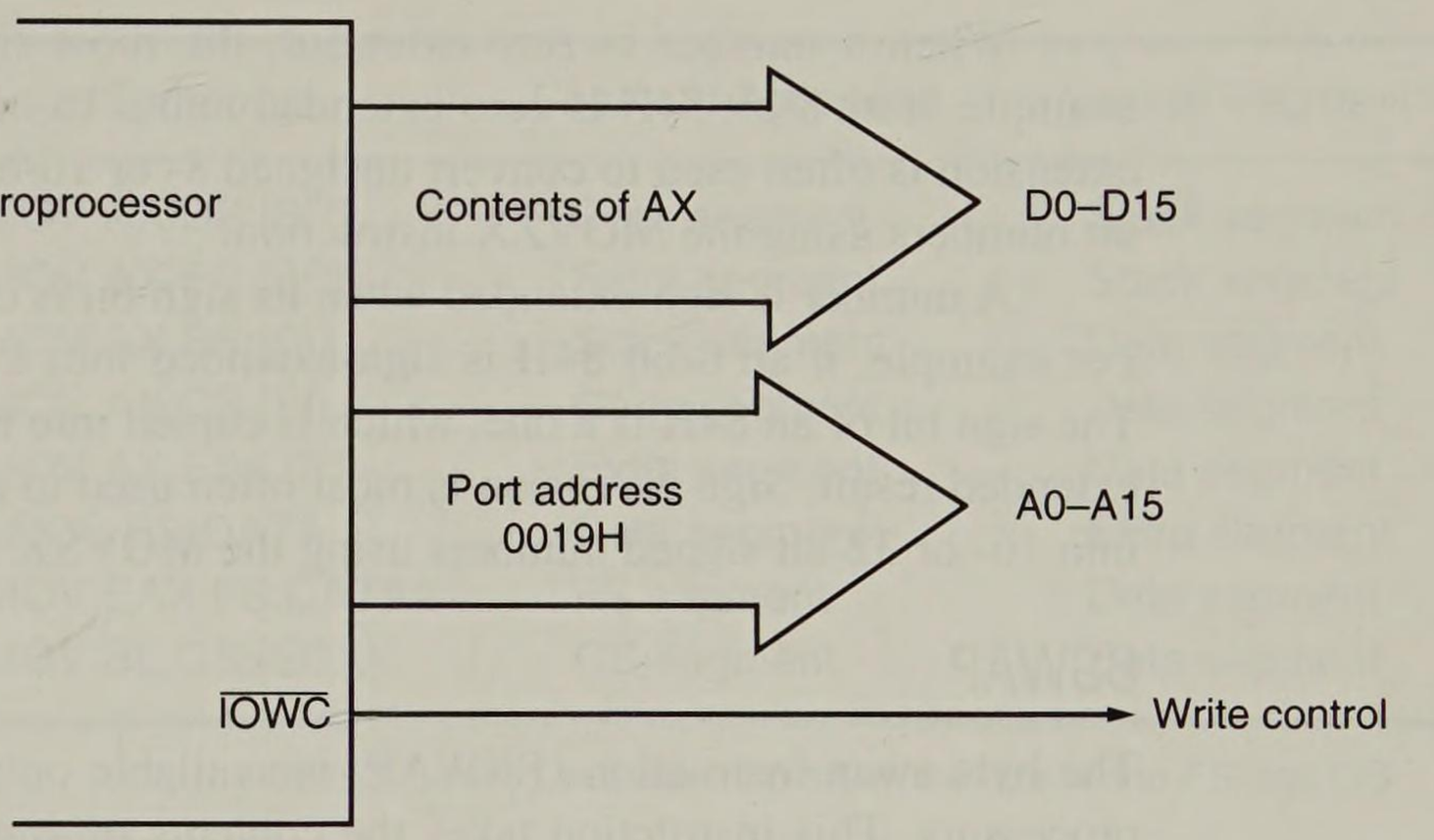



**FIGURE 4–18** The location of the port address, data, and write control signal for the microprocessor. The OUT 19H.AX instruction places a 0019H on the address bus and the contents of AX on the data bus.

## MOVSX and MOVZX

The MOVSX (move and sign-extend) and MOVZX (move and zero-extend) instructions are found in the 80386, 80486, and Pentium instruction sets. These instructions move data and at the same time either sign- or zero-extend it. Table 4–17 illustrates these instructions with several examples of each.

**TABLE 4–17** The MOVSX and MOVZX instructions

| *Symbolic* | *Example* | *Note* |
|---|---|---|
| MOVSX reg,reg | MOVSX CX,BL | Converts the 8-bit contents of BL into a 16-bit number in CX by sign-extension |
| | MOVSX ECX,AX | Converts the 16-bit contents of AX into a 32-bit number in ECX by sign-extension |
| MOVSX reg,mem | MOVSX BX,DATA | Converts the 8-bit contents of DATA into a 16-bit number in BX by sign-extension |
| | MOVSX EAX,[EDI] | Converts the 16-bit contents of the data segment memory location addressed by EDI into a 32-bit number in EAX by sign-extension |
| MOVZX reg,reg | MOVZX DX,AL | Converts the 8-bit contents of AL into a 16-bit number in DX by zero-extension |
| | MOVZX EBP,DI | Converts the 16-bit contents of DI into a 32-bit number in EBP by zero-extension |
| MOVZX reg,mem | MOVZX DX,DATA1 | Converts the 8-bit contents of DATA1 into a 16-bit number in DX by zero-extension |
| MOVZX reg,mem | MOVZX EAX,DATA2 | Converts the 16-bit contents of DATA2 into a 32-bit number in EAX by zero-extension |

When a number is zero-extended, the most significant part fills with zeros. For example, if an 8-bit 34H is zero-extended into a 16-bit number, it becomes 0034H. Zero-extension is often used to convert unsigned 8- or 16-bit numbers into unsigned 16- or 32-bit numbers using the MOVZX instruction.

A number is sign-extended when its sign bit is copied into the most significant part. For example, if an 8-bit 84H is sign-extended into a 16-bit number, it becomes FF84H. The sign bit of an 84H is a one, which is copied into the most significant part of the sign-extended result. Sign extension is most often used to convert 8- or 16-bit signed numbers into 16- or 32-bit signed numbers using the MOVSX instruction.

### BSWAP

The byte swap instruction (BSWAP) is available only in the 80486 and Pentium microprocessors. This instruction takes the contents of any 32-bit register and swaps the first byte with the fourth and the second with the third. For example, the BSWAP EAX instruction with EAX = 00112233H swaps bytes in EAX so it results in EAX = 33221100H. Notice that the order of all four bytes is reversed by this instruction.

## 4–6 SEGMENT OVERRIDE PREFIX

The segment override prefix, which may be added to almost any instruction in any memory addressing mode, allows the programmer to deviate from the default segment. The segment override prefix is an additional byte that appends the front of an instruction to select an alternate segment register. The only instructions that cannot be prefixed are the jump and call instructions, which must use the code segment register for address generation. The segment override is also used to select the FS and GS segments in the 80386, 80486, and Pentium microprocessors.

For example, the MOV AX,[DI] instruction accesses data within the data segment by default. If required by a program, this can be changed by prefixing the instruction. Suppose that the data are in the extra segment instead of the data segment. This instruction addresses the extra segment if changed to MOV AX,ES:[DI].

Table 4–18 shows some altered instructions that address a different memory segment than normal. Each time we prefix an instruction with a segment override prefix, the instruction becomes one byte longer. Although this is no serious change to the length of the instruction, it does add to its execution time. Usually, we limit the use of the segment override prefix in order to write shorter and more efficient software.

## 4–7 ASSEMBLER DETAIL

The assembler* for the 8086–80486 and Pentium microprocessors can be used in two ways: (1) with models that are unique to a particular assembler and (2) with full-segment

*The assembler used in this text is the Microsoft macro assembler, called MASM.

**TABLE 4–18** Instructions that include segment override prefixes

| *Symbolic* | *Segment Accessed* | *Normal Segment* |
|---|---|---|
| MOV AX,DS:[BP] | Data segment | Stack segment |
| MOV AX,ES:[BP] | Extra segment | Stack segment |
| MOV AX,SS:[DI] | Stack segment | Data segment |
| MOV AX,CS:[SI] | Code segment | Data segment |
| MOV AX,ES:LIST | Extra segment | Data segment |
| LODS ES:DATA | Data segment | Extra segment |
| MOV EAX,FS:DATA2 | FS segment | Data segment |
| MOV BL,GS:[ECX] | GS segment | Data segment |

*Note*: Only the 80386, 80486, and Pentium allow the use of the FS and GS segments.

definitions that allow complete control over the assembly process and are universal to all assemblers. This section of the text presents both methods and explains how to organize a program's memory space using the assembler. It also explains the purpose and use of some of the more important directives used with this assembler. Appendix A provides additional detail about the assembler.

## Directives

Before the format of an assembly language program is discussed, some details about the directives (*pseudo-operations*) that control the assembly process must be learned. Some common assembly language directives appear in Table 4–19. Directives indicate how an operand or section of a program is to be processed by the assembler. Some directives generate and store information in the memory, while others do not. The DB (define byte) directive stores bytes of data in the memory, while the BYTE PTR directive never stores data. The BYTE PTR directive indicates the size of the data referenced by a pointer or index register.

Note that the assembler, by default, accepts only 8086/8088 instructions unless a program is preceded by the .386 or .386P directive or one of the other microprocessor selection switches. The .386 directive tells the assembler to use the 80386 instruction set in the real mode, while the .386P directive tells the assembler to use the 80386 instruction set in the protected mode.

*Storing Data in a Memory Segment.* The DB (define byte), DW (define word), and DD (define doubleword) directives are most often used with the 8086–80486 and Pentium to define and store memory data. If a numeric coprocessor is present in the system, the DQ (define quadword) and DT (define ten bytes) directives are also common. These directives use the label to identify a memory location and the directive to indicate the size of the location.

**EXAMPLE 4–9** (pp. 121–123)

```
                        ;Using the DB, DW, and DD directives
                        ;
0000                    LIST_SEG    SEGMENT
```

**TABLE 4–19** Common assembler directives

| *Directive* | *Function* |
|---|---|
| .286 | Selects the 80286 instruction set |
| .286P | Selects the protected-mode 80286 instruction set |
| .386 | Selects the 80386 instruction set |
| .386P | Selects the protected-mode 80386 instruction set |
| .486 | Selects the 80486 instruction set |
| .486P | Selects the 80486 protected-mode instruction set |
| .287 | Selects the 80287 numeric coprocessor |
| .387 | Selects the 80387 numeric coprocessor |
| ALIGN 2 | Starts the data in a segment at word or doubleword boundaries |
| ASSUME | Indicates the names of each segment to the assembler; does not load the segment registers |
| AT | Indicates what physical segment address is used with the SEGMENT statement |
| BYTE | Indicates a byte-sized operand, as in BYTE PTR or THIS BYTE |
| DB | Defines byte(s) (8-bits) |
| DD | Defines doubleword(s) (32-bits) |
| DQ | Defines quadword(s) (64-bits) |
| DT | Defines ten bytes (80-bits) |
| DUP | Generates duplicates of characters or numbers |
| DW | Defines word(s) (16-bits) |
| DWORD | Indicates a doubleword-sized operand, as in THIS DWORD |
| END | Indicates the end of the program |

```
0000 01 02 03          DATA1  DB   1,2,3              ;define bytes
0003 45                       DB   45H                ;hexadecimal
0004 41                       DB   'A'                ;ASCII
0005 F0                       DB   11110000B          ;binary
0006 000C 000D         DATA2  DW   12,13              ;define words
000A 0200                     DW   LIST1              ;symbolic
000C 2345                     DW   2345H              ;hexadecimal
000E 00000300          DATA3  DD   300H               ;hexadecimal
0012 4007DF3B                 DD   2.123              ;real
0016 544269E1                 DD   3.34E+12           ;real
001A 00                LISTA  DB   ?                  ;reserve 1 byte
001B 000A[             LISTB  DB   10 DUP (?)         ;reserve 10 bytes
          ??
            ]
0025 00                ALIGN  2                       ;set word boundary

0026 0100[             LISTC  DW   100H DUP (0)       ;word array
          0000
              ]
0226 0016[             LIST_9 DD   22 DUP (?)         ;doubleword array
          ????????
```

**TABLE 4–19** *(continued)*

| *Directive* | *Function* |
|---|---|
| ENDM | Indicates the end of a macro sequence |
| ENDP | Indicates the end of a procedure |
| ENDS | Indicates the end of a segment or structure |
| EQU | Equates data to a label |
| FAR | Specifies a far address as in JMP FAR PTR LISTS |
| MACRO | Defines the name, parameters, and start of a macro |
| NEAR | Specifies a near address as in JMP NEAR PTR HELP |
| OFFSET | Specifies an offset address |
| ORG | Sets the origin within a segment |
| PROC | Defines the beginning of a procedure |
| PTR | Indicates a memory pointer |
| SEGMENT | Defines the start of a memory segment |
| STACK | Indicates that a segment is a stack segment |
| STRUC | Defines the start of a data structure |
| THIS | Used with EQU to set a label to a byte, word, or doubleword |
| USES | An MASM version 6.X directive that automatically saves registers used by a procedure |
| USE16 | Directs the assembler to use the 16-bit instruction mode and data sizes for the 80386, 80486, and Pentium microprocessors |
| USE32 | Directs the assembler to use the 32-bit instruction mode and data sizes for the 80386, 80486, and Pentium microprocessors |
| WORD | Acts as a word operand as in WORD PTR or THIS WORD |

*Note*: Most of these directives function with most versions of the assembler.

```
                        ]
027E 0064[              SIXES  DB   100 DUP (6)      ;byte array
          06
             ]
02E2                    LIST_SEG    ENDS
```

Example 4–9 shows a memory segment that contains various forms of data definition directives. The first statement (SEGMENT) indicates the start of the segment and its symbolic name. The last statement of the segment contains the ENDS directive that indicates the end of the segment. The name of the segment (LIST_SEG), can be anything that the programmer desires to call it. This allows a program to contain as many segments as required.

This example shows various forms of data storage for bytes at DATA1. More than one byte can be defined on a line in binary, hexadecimal, decimal, or ASCII code. The DATA2 label shows how to store various forms of word data. Doublewords are stored at DATA3 and they include floating-point single-precision real numbers.

Memory can be reserved for use in the future by using a ? as an operand for a DB, DW, or DD directive. When a ? is used in place of a numeric or ASCII value, the assembler sets aside a location and does not initialize it to any specific value. The DUP (*duplicate*) directive creates an array as shown in Example 4–9. A 10 DUP (?) reserves 10 locations of memory, but stores no specific value in any of the 10 locations. If a number appears within the ( ) part of the DUP statement, the assembler initializes the reserved section of memory with the data indicated.

The ALIGN directive, used in this example, makes sure that the memory arrays are stored on word boundaries. An ALIGN 2 places data on word boundaries for an 80286–80486 and Pentium microprocessor, and an ALIGN 4 places data on doubleword boundaries for an 80386, 80486, or Pentium. It is important that word-sized data is placed at word boundaries and doubleword-sized data at doubleword boundaries. If not, the microprocessor spends additional time accessing these data types. A word stored at an odd-numbered memory location can take twice as long to access as a word stored at an even-numbered memory location.

*EQU and THIS.* The equate (EQU) directive equates a numeric, ASCII, or label to another label. Equates make a program clearer and simplify debugging. Example 4–10 shows several equate statements and a few instructions that show how they function in a program.

**EXAMPLE 4–10**

```
                        ;Using equate directive
                        ;
= 000A                  TEN     EQU  10
= 0009                  NINE    EQU  9

0000 B0 0A                      MOV  AL,TEN
0002 04 09                      ADD  AL,NINE
```

The THIS directive always appears as THIS BYTE, THIS WORD, or THIS DWORD. In certain cases, data must be referred to as both a byte and a word. The assembler can only assign either a byte or a word address to a label. To assign a byte label to a word, use the software listed in Example 4–11.

**EXAMPLE 4–11**

```
                        ;Using the THIS and ORG directives
                        ;
0000                    DATA_SEG    SEGMENT
0100                            ORG  100H

= 0100                  DATA1   EQU  THIS BYTE
0100 0000               DATA2   DW   ?

0102                    DATA_SEG    ENDS

0000                    CODE_SEG    SEGMENT 'CODE'

                                ASSUME  CS:CODE_SEG,DS:DATA_SEG

0000 8A 1E 0100 R               MOV  BL,DATA1
0004 A1 0100 R                  MOV  AX,DATA2
```

```
0007 8A 3E 0101 R           MOV   BH,DATA1+1

000B                CODE_SEG    ENDS
```

This example also illustrates how the ORG (origin) statement changes the starting offset address of the data in the data segment to location 100H. The ASSUME statement tells the assembler what names have been chosen for the code, data, extra, and stack segments. Without the assume statement, the assembler assumes *nothing* and automatically uses a segment override prefix on all instructions that address memory data.

*PROC and ENDP.* The PROC and ENDP directives indicate the start and end of a procedure (*subroutine*). These directives force structure because the procedure is clearly defined. Both directives require a label to indicate the name of the subroutine. The PROC directive, which indicates the start of a procedure, must also be followed with either NEAR or FAR. A NEAR procedure is one that resides in the same code segment as the program. A FAR procedure may reside at any location in the memory system. We often call NEAR procedures *local*, and FAR procedures *global*. The term *global* denotes a procedure that can be used by any program, while *local* defines a procedure that is only used by the current program.

**EXAMPLE 4–12**

```
                    ;A procedure that adds BX, CX, and DX with the sum
                    ;stored in AX
                    ;
0000                ADDEM   PROC FAR            ;start procedure

0000 03 D9                  ADD   BX,CX
0002 03 DA                  ADD   BX,DX
0004 8B C3                  MOV   AX,BX
0006 CB                     RET

0007                ADDEM   ENDP                ;end procedure
```

Example 4–12 shows a procedure that adds BX, CX, and DX and stores the sum in register AX. Although this procedure is short and may not be particularly useful, it illustrates how to use the PROC and ENDP directives to delineate the procedure.

If Microsoft MASM version 6.X is available, the PROC directive specifies and automatically saves any registers used within the procedure. The *USES* statement indicates which registers are used by the procedure, so the assembler can automatically save them before the procedure begins and restore them before the procedure ends with the RET instruction. For example, the ADDS PROC USES AX BX CX statement automatically pushes AX, BX, and CX on the stack before the procedure begins and pops them from the stack before the RET instruction executes at the end of the procedure. Example 4–13 illustrates a procedure that shows the USES statement, written using MASM 6.X. Note that the registers in the list are separated not by commas, but spaces, and the PUSH and POP instructions are not displayed in the procedure listing. The opcode for the return instruction, although present, is also not displayed. The USES statement does not appear elsewhere in this text so compatibility with MASM version 5.10 can be maintained.

**EXAMPLE 4–13**

```
                        ;A procedure that includes the USES directive to save
                        ;BX, CX, and DX on the stack and restore them before
                        ;the RET instruction.
                        ;
0000                    ADDS    PROC NEAR  USES BX CX DX

0003 03 D8                      ADD  BX,AX
0005 03 CB                      ADD  CX,BX
0007 03 D1                      ADD  DX,CX
0009 8B C2                      MOV  AX,DX
                                RET

000F                    ADDS    ENDP
```

## Memory Organization

The assembler uses two basic formats for developing software. One method uses models and the other uses full-segment definitions. Memory models, as presented in this section, are unique to the MASM assembler program. The TASM assembler also uses memory models, but they differ somewhat from the MASM models. The full-segment definitions are common to most assemblers, including the *Intel assembler*, and are most often used for software development. The models are easier to use for simple tasks. The full-segment definitions offer better control over the assembly language task and are recommended and used in other places in this text. We use full-segment definitions because they apply to all assemblers, whereas models do not.

*Models.* There are many models that can be used with the MASM assembler, from tiny to huge. Appendix A lists all the models available for use with the assembler. To designate a model, we use the .MODEL statement followed by the size of the memory system. The tiny model, useful for small programs, requires that all software and data fit into one 64K byte memory segment. The small model requires that only one data segment is used with one code segment, for a total of 128K bytes of memory. Other models are available up to the huge model.

**EXAMPLE 4–14**

```
                        .MODEL SMALL
                        .STACK 100H                     ;define stack
                        .DATA                           ;define data segment

0000 0064[              LISTA  DB   100 DUP (?)
        ??
           ]
0064 0064[              LISTB  DB   100 DUP (?)
        ??
           ]

                        .CODE                           ;define code segment

0000 B8 ---- R          HERE:  MOV  AX,@DATA            ;load ES, DS
0003 8E C0                     MOV  ES,AX
```

```
0005 8E D8                      MOV  DS,AX

0007 FC                         CLD                          ;move data
0008 BE 0000 R                  MOV  SI,OFFSET LISTA
000B BF 0064 R                  MOV  DI,OFFSET LISTB
000E B9 0064                    MOV  CX,100
0011 F3/A4                      REP  MOVSB

0013                            .EXIT  0                     ;exit to DOS

                                END  HERE
```

Example 4–14 illustrates how the .MODEL statement defines the parameters of a short program that copies the contents of a 100-byte block of memory (LISTA) into a second 100-byte block of memory (LISTB). It also shows how to define the stack, data, and code segments. The .EXIT 0 directive returns to DOS with an error code of 0 (no error). Also note that special directives such as @DATA (see Appendix A) are used to identify various segments. Models are important with both Microsoft C/C++ and Borland C/C++ development systems if assembly language is included with C/C++ programs. Both development systems use in-line assembly programming for adding assembly language instructions and require an understanding of programming models. Refer to the respective C/C++ language reference for each system to determine the model protocols.

*Full Segment Definitions.* Example 4–15 illustrates the same program as Example 4–14 using full-segment definitions. Full-segment definition is also used with the Borland and Microsoft C/C++ environments for procedures developed in assembly language. The program in Example 4–15 is longer and more structured than the one pictured in Example 4–14. The first segment defined is the STACK_SEG, which is clearly delineated with the SEGMENT and ENDS directives. Within these directives a DW 100 DUP (?) sets aside 100H words for the stack segment. Because the word STACK appears next to SEGMENT, the assembler and linker automatically load both the stack segment register (SS) and stack pointer (SP).

**EXAMPLE 4–15**

```
0000                    STACK_SEG   SEGMENT  STACK

0000 0100[                          DW   100H DUP (?)
         ????
                 ]
0200                    STACK_SEG   ENDS

0000                    DATA_SEG    SEGMENT  'DATA'

0000 0064[              LISTA  DB   100 DUP (?)
         ??
           ]
0064 0064[              LISTB  DB   100 DUP (?)
         ??
             ]

00C8                    DATA_SEG    ENDS
```

```
0000                    CODE_SEG    SEGMENT 'CODE'
                                    ASSUME CS:CODE_SEG,DS:DATA_SEG
                                    ASSUME SS:STACK_SEG

0000                    MAIN    PROC FAR

0000 B8 ---- R                  MOV  AX,DATA_SEG      ;load DS and ES
0003 8E C0                      MOV  ES,AX
0005 8E D8                      MOV  DS,AX

0007 FC                         CLD                   ;move data
0008 BE 0000 R                  MOV  SI,OFFSET LISTA
000B BF 0064 R                  MOV  DI,OFFSET LISTB
000E B9 0064                    MOV  CX,100
0011 F3/A4                      REP  MOVSB

0013 B4 4C                      MOV  AH,4CH           ;exit to DOS
0015 CD 21                      INT  21H

0017                    MAIN    ENDP

0017                    CODE_SEG    ENDS
                                    END  MAIN
```

Next, the data are defined in the DATA_SEG. Here two arrays of data appear as LISTA and LISTB. Each array contains 100 bytes of space for the program. The names of the segments in this program can be changed to any name. We include the group name 'DATA' so the Microsoft program CODEVIEW can be effectively used to symbolically debug this software. If the group name is not placed in a program, CODEVIEW can still be used to debug a program, but the program will not be debugged in symbolic form. Other group names such as 'STACK', 'CODE', and so forth are listed in Appendix A.

The CODE_SEG is organized as a far procedure, because most software is procedure oriented. Before the program begins, the code segment contains the ASSUME statement. The ASSUME statement tells the assembler and linker the name used for the code segment (CS) is CODE_SEG. It also tells the assembler and linker that the data segment is DATA_SEG and the stack segment is STACK_SEG. We include the group name 'CODE' for the code segment for use by CODEVIEW. Other group names appear in Appendix A.

After the program loads both the extra segment register and data segment register with the location of the data segment, it transfers 100 bytes from LISTA to LISTB. Following this is a sequence of two instructions that return control to DOS (the disk operating system).

The last statement in the program is END MAIN. The END statement indicates the end of the program and the location of the first instruction executed. Here we want the machine to execute the main procedure so a label follows the END directive.

In the 80386/80486/Pentium microprocessor an additional directive is attached to the code segment. The USE16 or USE32 directive tells the assembler to use either the 16- or 32-bit instruction modes for the microprocessor. Software developed for the DOS environment must use the USE16 directive for the 80386/80486/Pentium program to function correctly, because MASM assumes that all segments are 32 bits and all instruction modes are 32 bits by default. In fact, any program designed to execute in the real mode must include the USE16 directive if we deviate from the default 8086/8088.

Example 4–16 shows how the same software listed in Example 4–15 is formed for the 80386 microprocessor.

**EXAMPLE 4–16**

```
                        .386                            ;select the 80386
0000                    STACK_SEG   SEGMENT STACK

0000 0100[                  DW   100H DUP (?)
          ????
                ]
0200                    STACK_SEG   ENDS

0000                    DATA_SEG    SEGMENT  'DATA'

0000 0064[              LISTA  DB   100 DUP (?)
          ??
             ]
0064 0064[              LISTB  DB   100 DUP (?)
          ??
             ]

00C8                    DATA_SEG    ENDS

0000                    CODE_SEG    SEGMENT  USE16 'CODE'
                              ASSUME CS:CODE_SEG,DS:DATA_SEG
                              ASSUME SS:STACK_SEG

0000                    MAIN   PROC FAR

0000 B8 ---- R                 MOV  AX,DATA_SEG      ;load DS and ES
0003 8E C0                     MOV  ES,AX
0005 8E D8                     MOV  DS,AX

0007 FC                        CLD                   ;move data
0008 BE 0000 R                 MOV  SI,OFFSET LISTA
000B BF 0064 R                 MOV  DI,OFFSET LISTB
000E B9 0064                   MOV  CX,100
0011 F3/A4                     REP  MOVSB

0013 B4 4C                     MOV  AH,4CH           ;exit to DOS
0015 CD 21                     INT  21H

0017                    MAIN   ENDP

0017                    CODE_SEG    ENDS
                              END  MAIN
```

## A Sample Program

Example 4–17 provides a sample program that reads a character from the keyboard and displays it on the CRT screen. Although this program is trivial, it illustrates a complete, workable program that functions on any personal computer using DOS, from the earliest 8088-based system to the latest Pentium-based system. This program also illustrates the use of a few DOS function calls. Appendix A lists the DOS function calls with their parameters. The BIOS function calls allow the use of the keyboard, printer, disk drives, and everything else that is available in your computer system.

**EXAMPLE 4–17**

```
                        ;A program that reads a key and displays it.
                        ;Note that an @ key ends the program
                        ;
0000                    CODE_SEG    SEGMENT  'CODE'

                                ASSUME CS:CODE_SEG

0000                    MAIN    PROC FAR
0000 B4 06                      MOV  AH,6               ;read key
0002 B2 FF                      MOV  DL,0FFH
0004 CD 21                      INT  21H
0006 74 F8                      JE   MAIN              ;if no key

0008 3C 40                      CMP  AL,'@'            ;test for @
000A 74 08                      JE   MAIN1             ;if @

000C B4 06                      MOV  AH,6              ;display key
000E 8A D0                      MOV  DL,AL
0010 CD 21                      INT  21H
0012 EB EC                      JMP  MAIN              ;repeat
0014                    MAIN1:
0014 B4 4C                      MOV  AH,4CH            ;exit to DOS
0016 CD 21                      INT  21H

0018                    MAIN    ENDP

0018                    CODE_SEG    ENDS
                            END  MAIN
```

This example program uses only a code segment because there is no data. A stack segment should appear, but has been left out because DOS automatically allocates a 128-byte stack for all programs. The only time the stack is used in this example is for the INT 21H instructions that call a procedure in DOS. Note that when this program is linked, the linker signals that no stack segment is present. This warning may be ignored in this example because the stack is less than 128 bytes.

The program uses DOS functions 06H and 4CH. The function number is placed in AH before the INT 21H instruction executes. The 06H function reads the keyboard if DL = 0FFH or displays the ASCII contents of DL if it is not 0FFH. The first section of the program moves a 06H into AH and a 0FFH into DL, so a key is read from the keyboard. The INT 21H tests the keyboard and if no key is typed, it returns with an equal condition. The JE instruction tests the equal condition and jumps to MAIN if no key is typed.

When a key is typed, the program continues to the next step. This step compares the contents of AL with an @ symbol. Upon return from the INT 21H, the ASCII character of the typed key is found in AL. In this program if we type an @ symbol, the program ends. If we do not type an @ symbol the program continues by displaying the character typed on the keyboard with the next INT 21H instruction.

The second INT 21H instruction moves the ASCII character into DL so it can be displayed on the CRT screen. After displaying the character a JMP executes. This causes the program to continue at MAIN where it repeats reading a key.

If the @ symbol is typed, the program continues at MAIN1 where it executes the DOS function code number 4CH. This causes the program to return to the DOS prompt (A>) so the computer can be used for other tasks.

More information about the assembler and its applications appears in Appendix A and in the next several chapters. The appendix provides a complete overview of the assembler, linker, and DOS functions. It also provides a list of the BIOS (*basic I/O system*) functions. The information provided in the following chapters clarifies how to use the assembler for certain tasks.

## 4–8 SUMMARY

1. Data movement instructions transfer data between registers, a register and memory, a register and the stack, memory and the stack, the accumulator and I/O, and the flags and the stack. Memory-to-memory transfers are only allowed with the MOVS instruction.
2. Data movement instructions include: MOV, PUSH, POP, XCHG, XLAT, IN, OUT, LEA, LDS, LES, LSS, LGS, LFS, LAHF, SAHF, and the string instructions: LODS, STOS, MOVS, INS, and OUTS.
3. The first byte of an instruction contains the opcode. The opcode specifies the operation performed by the microprocessor. The opcode may be preceded one or more override prefixes in some forms of instructions.
4. The D bit, located in many instructions, selects the direction of data flow. If D = 1, the data flow from the REG field to the R/M field of the instruction. If D = 1, the data flow from the R/M field to the REG field.
5. The W bit, found in most instructions, selects the size of the data transfer. If W = 0, the data are byte-sized and if W = 1, the data are word-sized. In the 80386/80486/Pentium microprocessor W = 1 can also specify a 32-bit register.
6. MOD selects the addressing mode of operation for a machine language instruction's R/M field. If MOD = 00, there is no displacement; if MOD = 01, an 8-bit sign-extended displacement appears; if MOD = 10, a 16-bit displacement occurs; and if MOD = 11, a register is used instead of a memory location. In the 80386/80486/Pentium the MOD bits also specify a 32-bit displacement.
7. A 3-bit binary register code specifies the REG and R/M fields when MOD = 11. The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL, and the 16-bit registers are AX, BX, CX, DX, SP, BP, DI, and SI. The 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI.
8. When the R/M field depicts a memory mode, a 3-bit code selects one of the following modes: [BX + DI], [BX + SI], [BP + DI], [BP + SI], [BX], [BP], [DI], or [SI] for 16-bit instruction in the 8086–80486 and Pentium. In the 80386/80486/Pentium, the R/M field specifies: EAX, EBX, ECX, EDX, EBP, EDI, and ESI, or one of the scaled-index modes of addressing memory data. If the scaled-index mode is selected (R/M = 100), an additional byte (scaled index byte) is added to the instruction to specify the base register, index register, and scaling factor.
9. All memory addressing modes, by default, address data in the data segment, unless BP or EBP addresses memory. The BP or EBP register addresses data in the stack segment.
10. The segment registers are only addressed by the MOV, PUSH, or POP instruction. The MOV instruction may transfer a segment register to a 16-bit register or vice versa.

We do not allow the MOV CS,reg or POP CS instruction. In the 80386, 80486, and Pentium microprocessors, two additional segment registers, FS and GS, exist.

11. Data are transferred between a register or a memory location and the stack by the PUSH and POP instructions. Variations of these instructions allow immediate data to be pushed onto the stack, the flags to be transferred to the stack, and all the 16-bit registers to be transferred between the stack and the registers. When data are transferred to the stack, two bytes (8086–80286) always move. The least-significant byte is placed at the SP location –1 byte and the most significant byte is placed at the SP location –2 bytes. After placing the data on the stack, SP decrements by 2. In the 80386/80486/Pentium, four bytes of data from a memory location or register may also be transferred to the stack.
12. Opcodes that transfer data between the stack and the flags are PUSHF and POPF. Opcodes that transfer all the 16-bit registers between the stack and the registers are PUSHA and POPA. In the 80386/80486/Pentium PUSHFD and POPFD transfer the contents of the EFLAGS.
13. LEA, LDS, and LES instructions load a register or registers with an effective address. The LEA instruction loads any 16-bit register with an effective address, while LDS and LES load any 16-bit register and either DS or ES with the effective address. In the 80386/80486/Pentium, additional instructions include: LFS, LGS, and LSS that load a 16-bit register and FS, GS, or SS.
14. String data transfer instructions use either or both DI and SI to address memory. The DI offset address is located in the extra segment; the SI offset address is located in the data segment.
15. The direction flag (D) chooses the auto-increment or auto-decrement mode of operation for DI and SI string instructions. If we clear D to 0 with the CLD instruction, we select the auto-increment mode; if we set D to 1 with STD, we select the auto-decrement mode. Either or both DI and SI increment/decrement by one for a byte operation and by two for a word operation.
16. LODS loads AL, AX, or EAX with data from the memory location addressed by SI; STOS stores AL, AX, or EAX in the memory location addressed by DI; and MOVS transfers a byte or a word from the memory location addressed by SI into the location addressed by DI.
17. INS inputs data from an I/O device addressed by DX and stores it in the memory location addressed by DI; OUTS outputs the contents of the memory location addressed by SI and sends it to the I/O device addressed by DX.
18. The REP prefix may be attached to any string instruction to repeat it. The REP prefix repeats the string instruction the number of times found in register CX.
19. Translate (XLAT) converts the data in AL into a number stored at the memory location addressed by BX plus AL.
20. IN and OUT transfer data between AL, AX, or EAX and an external I/O device. The address of the I/O device is either stored with the instruction (fixed-port address) or in register DX (variable-port address).
21. The segment override prefix selects a segment register other than the default segment for a memory location. For example, the MOV AX,[BX] instruction uses the data segment, but the MOV AX,ES:[BX] instruction uses the extra segment because of the

ES: prefix. The segment override prefix is the only way the FS and GS segments can be addressed in the 80386/80486/Pentium microprocessor.

22. The MOVZX (move and zero-extend) and MOVSX (move and sign-extend) instructions found in the 80386/80486/Pentium microprocessor increase the size of a byte to a word or the size of a word to a doubleword. MOVSX increases the size of the number by inserting leading zeros. The MOVSX increases the size of the number by copying the sign bit into the more significant bits of the number.
23. Assembler directives DB, (define byte), DW (define word), DD (define doubleword), and DUP (duplicate) store data in the memory system.
24. The EQU (equate) directive allows data or labels to be equated to labels.
25. The SEGMENT directive identifies the start of a memory segment and ENDS identifies the end of a memory segment.
26. The ASSUME directive tells the assembler what segment names have been assigned to CS, DS, ES, and SS. In the 80386/80486/Pentium, ASSUME also indicates the segment name for FS and GS.
27. The PROC and ENDP directives indicate the start and end of a procedure. The USES directive (MASM version 6.X) automatically saves and restores any number of registers on the stack if it appears with the PROC directive.
28. The assembler assumes that software is being developed for the 8086/8088 microprocessor unless the .286, .386, or .486 directive is used to select one of these other microprocessors.
28. Memory models can be used to shorten the program slightly, but they are difficult to use for larger programs. Memory models are not compatible with all assembler programs.

## 4–9 QUESTIONS AND PROBLEMS

1. The first byte of an instruction is the _____ unless it contains one of the override prefixes.
2. Describe the purpose of the D and W bits found in some machine language instructions.
3. In a machine language instruction, the MOD field specifies what information?
4. If the register field (REG) of an instruction contains a 010 and W = 0, what register is selected? Assume that the instruction is a 16-bit mode instruction.
5. How are the 32-bit registers selected for the 80386/80486/Pentium microprocessor?
6. What memory addressing mode is specified by R/M = 001 with MOD = 00 for a 16-bit instruction?
7. Identify the default segment register assigned to:
   (a) SP
   (b) EBX
   (c) DI
   (d) EBP
   (e) SI
8. Convert 8B07H from machine language to assembly language.

9. Convert 8B1E004CH from machine language to assembly language.
10. If a MOV SI,[BX + 2] instruction appears in a program, what is its machine language equivalent?
11. If a MOV ESI,[EAX] instruction appears in a program for the 80386/80486/Pentium microprocessor operated in the 16-bit instruction mode, what is its machine language equivalent?
12. What is wrong with a MOV CS,AX instruction?
13. Form a short sequence of instructions that load the data segment register with a 1000H.
14. The PUSH and POP instructions always transfer a _____-bit number between the stack and a register or memory location in the 8086–80286 microprocessor.
15. What segment register may not be popped from the stack?
16. Which registers move onto the stack with a PUSHA instruction?
17. Which registers move onto the stack with a PUSHAD instruction?
18. Describe the operation of each of the following instructions:
    (a) PUSH AX
    (b) POP ESI
    (c) PUSH [BX]
    (d) PUSHFD
    (e) POP DS
    (f) PUSHD 4
19. Explain what happens when the PUSH BX instruction executes. Make sure to show where BH and BL are stored. (Assume that SP = 0100H and SS = 0200H.)
20. Repeat Question 19 for the PUSH EAX instruction.
21. The 16-bit POP instruction (except for POPA) increments SP by _____.
22. What values appear in SP and SS if the stack is addressed at memory location 02200H?
23. Compare the operation of a MOV DI,NUMB instruction with an LEA DI,NUMB instruction.
24. What is the difference between an LEA SI,NUMB instruction and a MOV SI,OFFSET NUMB instruction?
25. Which is more efficient, a MOV with an OFFSET, or an LEA instruction?
26. Describe how the LDS BX,NUMB instruction operates.
27. What is the difference between the LDS and LSS instructions?
28. Develop a sequence of instructions that move the contents of data segment memory locations NUMB and NUMB+1 into BX, DX, and SI.
29. What is the purpose of the direction flag?
30. Which instructions set and clear the direction flag?
31. The string instructions use DI and SI to address memory data in which memory segments?
32. Explain the operation of the LODSB instruction.
33. Explain the operation of the STOSW instruction.
34. Explain the operation of the OUTSB instruction.
35. What does the REP prefix accomplish and with what type of instruction is it used?
36. Develop a sequence of instructions that copy 12 bytes of data from an area of memory addressed by SOURCE into an area of memory addressed by DEST.

37. Where is the I/O address (port number) stored for an INSB instruction?
38. Select an assembly language instruction that exchanges the contents of the EBX register with the ESI register.
39. Would the LAHF and SAHF instructions normally appear in software?
40. Explain how the XLAT instruction transforms the contents of the AL register.
41. Write a short program that uses the XLAT instruction to convert the BCD numbers 0–9 into ASCII-coded numbers 30H–39H. Store the ASCII-coded data in a TABLE located within the data segment.
42. Explain what the IN AL,12H instruction accomplishes.
43. Explain how the OUT DX,AX instruction operates.
44. What is a segment override prefix?
45. Select an instruction that moves a byte of data from the memory location addressed by the BX register in the extra segment into the AH register.
46. Develop a sequence of instructions to exchange the contents of AX with BX, ECX with EDX, and SI with DI.
47. What is an assembly language directive?
48. Describe the purpose of the following assembly language directives: DB, DW, and DD.
49. Select an assembly language directive that reserves 30 bytes of memory for array LIST1.
50. Describe the purpose of the EQU directive.
51. What is the purpose of the .386 directive?
52. What is the purpose of the .MODEL directive?
53. If the start of a segment is identified with .DATA, what type of memory organization is in effect?
54. If the SEGMENT directive identifies the start of a segment, what type of memory organization is in effect?
55. What does the INT 21H accomplish if AH contains a 4CH?
56. What directives indicate the start and end of a procedure?
57. Explain the purpose of the USES statement as it applies to a procedure with MASM version 6.X.
58. How is the 80486 microprocessor instructed to use the 16-bit instruction mode?
59. Develop a near procedure that stores AL in four consecutive memory locations within the data segment, as addressed by the DI register.
60. Develop a far procedure that copies contents of the word-sized memory location CS:DATA1 into AX, BX, CX, DX, and SI.

# CHAPTER 5

# Arithmetic and Logic Instructions

## INTRODUCTION

In this chapter, we examine the arithmetic and logic instructions found in the 8086–80486 and Pentium instruction sets. Arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, incrementation, and decrementation. Logic instructions include: AND, OR, Exclusive-OR, NOT, shifts, rotates, and the logical compare (TEST). Also presented are the 80386/80486/Pentium instructions: SHRD, SHLD, bit tests, and bit scans.

We also will introduce string comparison instructions, which are used for scanning tabular data and for comparing sections of memory data. Both tasks perform efficiently with the string scan (SCAS) and string compare (CMPS) instructions.

If you are already familiar with an 8-bit microprocessor, you will recognize that the 8086–80486 and Pentium instruction set is superior because most of the instructions have two operands instead of one. Even if this is your first microprocessor, you will quickly learn that the microprocessor possesses a powerful and easy-to-use set of arithmetic and logic instructions.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use the arithmetic and logic instructions to accomplish simple binary, BCD, and ASCII arithmetic.
2. Use AND, OR, and Exclusive-OR to accomplish binary bit manipulation.
3. Use the shift and rotate instructions.
4. Explain the operation of the 80386/80486/Pentium double-precision shift, bit test, and bit scan instructions.
5. Check the contents of a table for a match with the string instructions.

# 5–1 ADDITION, SUBTRACTION, AND COMPARISON

The bulk of the arithmetic instructions found in any microprocessor includes addition, subtraction, and comparison. The 8086–80486 and Pentium microprocessors are no different. In this section we illustrate and define these instructions. We also show their use in manipulating register and memory data.

## Addition

Addition appears in many forms in the 8086–80486 and Pentium microprocessors. This section details the use of the ADD instruction for 8-, 16-, and 32-bit binary addition. Another form of addition, called *add-with-carry,* is introduced with the ADC instruction. Finally, the increment instruction (INC), a special type of addition that adds one, is presented. In Section 5–3 we examine other forms of addition, such as BCD and ASCII.

Table 5–1 illustrates the addressing modes available to the ADD instruction. (These addressing modes include almost all those mentioned in Chapter 3.) Since there are over 1,000 variations of the ADD instruction in the instruction set, it is impossible to list them all in this table. The only types of addition not allowed are memory-to-memory and segment register. The segment registers can only be moved, pushed, or popped. As with all other instructions, the 32-bit registers are only available with the 80386/80486/Pentium microprocessor.

**EXAMPLE 5–1**

```
0000                    ADDS    PROC NEAR

0000 03 C3                      ADD  AX,BX
0002 03 C1                      ADD  AX,CX
0004 03 C2                      ADD  AX,DX
0006 C3                         RET

0007                    ADDS    ENDP
```

*Register Addition.* Example 5–1 shows a simple procedure that uses register addition to add several registers. In this example, we add the contents of AX, BX, CX, and DX to form a 16-bit result stored in the AX register. Here we use a procedure, because assembly language is procedure-oriented.

Whenever arithmetic and logic instructions execute, the contents of the flag register change. Note that the contents of the interrupt, trap, and other flags do not change due to arithmetic and logic instructions. Only the flags located in the right-most eight bits of the flag register and the overflow flag change. These right-most flags denote the result of an arithmetic or logic operation. Any ADD instruction modifies the contents of the sign, zero, carry, auxiliary carry, parity, and overflow flags. The flag bits never change for most of the data transfer instructions presented in Chapter 4.

**EXAMPLE 5–2**

```
0006 B2 12                      MOV  DL,12H
0008 80 C2 33                   ADD  DL,33H
```

**TABLE 5–1** Addition instructions

| *Instruction* | *Comment* |
|---|---|
| ADD AL,BL | AL = AL + BL |
| ADD CX,DI | CX = CX + DI |
| ADD EBP,EAX | EBP = EBP + EAX |
| ADD CL,44H | CL = CL + 44H |
| ADD BX,35AFH | BX = BX + 35AFH |
| ADD EDX,12345H | EDX = EBX + 00012345H |
| ADD [BX],AL | AL adds to the contents of the data-segment offset location addressed by BX, and the result is stored in the same memory location |
| ADD CL,[BP] | The contents of the stack-segment offset location addressed by BP add to CL, and the result is stored in CL |
| ADD AL,[EBX] | The contents of the data-segment offset location addressed by EBX add to AL, and the result is stored in AL |
| ADD BX,[SI + 2] | The word-sized contents of the data-segment location addressed by SI plus 2 add to BX, and the result is stored in the same memory location |
| ADD CL,TEMP | The contents of data-segment location TEMP add to CL, with the result stored in CL |
| ADD BX,TEMP[DI] | The word-sized contents of the data-segment location addressed by TEMP plus DI add to BX and the result is stored in the same memory location |
| ADD [BX + DI],DL | The data-segment memory byte addressed by BX + DI is the sum of that byte plus DL |
| ADD BYTE PTR [DI],3 | Add a 3 to the contents of the byte-sized memory location addressed by DI within the data segment |
| ADD BX,[EAX+2*ECX] | The data-segment memory word, addressed by the sum of 2 times ECX plus EAX, adds to BX |

*Immediate Addition.* Immediate addition is employed whenever constant or known data are added. An 8-bit immediate addition appears in Example 5–2. In this example, we first load DL with a 12H by using an immediate move instruction. Next, we add a 33H to the 12H in DL using an immediate addition instruction. After the addition, the sum (45H) moves into register DL and the flags change as follows:

Z = 0 (result not zero)

C = 0 (no carry)

A = 0 (no half-carry)

S = 0 (result positive)

P = 0 (odd parity)

O = 0 (no overflow)

*Memory-to-Register Addition.* Suppose an application requires that memory data add to the AL register. Example 5–3 shows an example that adds two consecutive bytes of data, stored at the data-segment offset locations NUMB and NUMB+1, to the AL register.

**EXAMPLE 5–3**

```
0000                    SUMS    PROC NEAR

0000 BF 0000 R                  MOV  DI,OFFSET NUMB  ;address NUMB
0003 B0 00                      MOV  AL,0            ;clear sum
0005 02 05                      ADD  AL,[DI]         ;add NUMB
0007 02 45 01                   ADD  AL,[DI+1]       ;add NUMB+1
000A C3                         RET

000B                    SUMS    ENDP
```

The procedure first loads the contents of the destination index register (DI) with offset address NUMB. The DI register, used in this example, addresses data in the data segment beginning at memory location NUMB. Next, the ADD AL,[DI] instruction adds the contents of memory location NUMB to AL. This occurs because DI addresses memory location NUMB, and the instruction adds its contents to AL. Finally, the ADD AL,[DI+1] instruction adds the contents of memory location NUMB plus one byte to the AL register. After both ADD instructions execute, the result appears in the AL register as the sum of the contents of NUMB plus the contents of NUMB+1.

**EXAMPLE 5–4**

```
0000                    SUM     PROC NEAR

0000 B0 00                      MOV  AL,0              ;clear sum
0002 BE 0003                    MOV  SI,3              ;address element 3
0005 02 84 0002 R               ADD  AL,ARRAY[SI]      ;add element 3
0009 02 84 0004 R               ADD  AL,ARRAY[SI+2]    ;add element 5
000D 02 84 0006 R               ADD  AL,ARRAY[SI+4]    ;add element 7
0011 C3                         RET

0012                    SUM     ENDP
```

*Array Addition.* Memory arrays are sequential lists of data. Suppose that an array of data (ARRAY) contains ten bytes numbered from element 0 through element 9. Example 5–4 shows a procedure that adds the contents of array elements 3, 5, and 7. (The procedure and the array elements it adds are chosen to demonstrate the use of some of the addressing modes for the microprocessor.)

This example first clears AL to zero, so it can be used to accumulate the sum. Next, we load register SI with a 3 to initially address array element 3. The ADD AL,ARRAY[SI] instruction adds the contents of array element 3 to the sum in AL. The instructions that follow add array elements 5 and 7 to the sum in AL, using a 3 in SI plus a displacement of 2 to address element 5, and a displacement of 4 to address element 7.

Suppose that an array of data contains 16-bit numbers used to form a 16-bit sum in register AX. Example 5–5 shows a procedure written for the 80386/80486/Pentium microprocessor using the scaled-index form of addressing to add elements 3, 5, and 7 of an area

of memory called ARRAY. In this example, we load EBX with the address ARRAY, and ECX holds the array element number. Note how the scaling factor is used to address words of data by multiplying the contents of the ECX register by two. Recall that words are two bytes in length.

**EXAMPLE 5–5**

```
0000                        SUM     PROC NEAR
0000 66| BB 00000000 R      MOV   EBX,OFFSET ARRAY  ;address ARRAY
0006 66| B9 00000003        MOV   ECX,3             ;address element 3
000C 67& 8B 04 4B           MOV   AX,[EBX+2*ECX]    ;get element 3
0010 66| B9 00000005        MOV   ECX,5             ;address element 5
0016 67& 03 04 4B           ADD   AX,[EBX+2*ECX]    ;add element 5
001A 66| B9 00000007        MOV   ECX,7             ;address element 7
0020 67& 03 04 4B           ADD   AX,[EBX+2*ECX]    ;add element 7
0024 C3                     RET

0025                    SUM     ENDP
```

*Increment Addition.* Increment addition (INC) adds one to a register or a memory location. The INC instruction can add one to any register or memory location except a segment register. Table 5–2 illustrates some of the possible forms of the increment instruction available to the 8086–80486 and Pentium microprocessors. It is impossible to show all variations of the INC instruction because of their large number.

With indirect memory increments, the size of the data must be described using the BYTE PTR, WORD PTR, or DWORD PTR directives. The reason is that the assembler program cannot determine whether, for example, the INC [DI] instruction is a byte-, word-, or doubleword-sized increment. The INC BYTE PTR [DI] instruction indicates byte-sized memory data; the INC WORD PTR [DI] instruction indicates word-sized memory data; and the INC DWORD PTR [DI] instruction indicates doubleword-sized data.

**TABLE 5–2** Increment instructions

| *Instruction* | *Comment* |
|---|---|
| INC BL | BL = BL + 1 |
| INC SP | SP = SP + 1 |
| INC EAX | EAX = EAX + 1 |
| INC BYTE PTR [BX] | The byte contents of the memory location addressed by BX in the data-segment increment |
| INC WORD PTR [SI] | The word contents of the memory location addressed by SI in the data-segment increment |
| INC DWORD PTR [ECX] | The doubleword contents of the data-segment memory location addressed by ECX increment |
| INC DATA1 | The contents of data-segment location DATA1 increment |

**EXAMPLE 5–6**

```
0000                    SUMS    PROC NEAR

0000 BF 0000 R                  MOV  DI,OFFSET NUMB  ;address NUMB
0003 B0 00                      MOV  AL,0            ;clear sum
0005 02 05                      ADD  AL,[DI]         ;add NUMB
0007 47                         INC  DI              ;address NUMB+1
0008 02 05                      ADD  AL,[DI]         ;add NUMB+1
000A C3                         RET

000B                    SUMS    ENDP
```

Example 5–6 shows how the procedure shown in Example 5–3 can be modified to use the increment instruction for addressing NUMB and NUMB+1. Here, an INC DI instruction changes the contents of register DI from offset address NUMB to offset address NUMB+1. Examples 5–3 and 5–6 both add the contents of NUMB and NUMB+1. The difference between these programs is the way the data address is formed through the contents of the DI register using the increment instruction.

Increment instructions affect the flag bits, as do most other arithmetic and logic operations. But increment instructions do not affect the carry flag bit. Carry doesn't change because we often use increments in programs that depend upon the contents of the carry flag.

*Addition with Carry.* An addition-with-carry instruction (ADC) adds the bit in the carry flag (C) to the operand data. This instruction mainly appears in software that adds numbers that are wider than 16 bits in the 8086–80286, or wider than 32 bits in the 80386/80486/Pentium microprocessor.

Table 5–3 lists several add-with-carry instructions with comments that explain their operation. Like the ADD instruction, ADC affects the flags after the addition.

**TABLE 5–3** Add-with-carry instructions

| *Instruction* | *Comment* |
|---|---|
| ADC AL,AH | AL = AL + AH + carry |
| ADC CX,BX | CX = CX + BX + carry |
| ADC EBX,EDX | EBX = EBX + EDX + carry |
| ADC [BX],DH | The byte contents of the data-segment memory location addressed by BX are added to DH and carry; the result is stored in memory |
| ADC BX,[BP + 2] | BX and the word contents of the stack-segment memory location addressed by BP are added to carry; the result is stored in BX |
| ADC ECX,[EBX] | ECX and the doubleword contents of the data-segment memory location addressed by EBX are added to carry; the result is stored in ECX |

*Note*: Only the 80386/80486/Pentium use 32-bit registers and addressing modes.

Suppose we write a program for the 8086–80286 to add the 32-bit number in BX and AX to the 32-bit number in DX and CX. Figure 5–1 illustrates this addition so the placement and function of the carry flag can be understood. This addition cannot be easily performed without adding the carry flag bit because the 8086–80286 only adds 8- or 16-bit numbers. Example 5–7 shows how the addition occurs with a procedure. Here the contents of registers AX and CX add to form the least-significant 16 bits of the sum. This addition may or may not generate a carry. A carry appears in the carry flag if the sum is greater than FFFFH. Because it is impossible to predict a carry, the most-significant 16 bits of this addition are added with the carry flag, using the ADC instruction. The ADC instruction adds the one or the zero in the carry flag to the most-significant 16 bits of the result. This program adds BX–AX to DX–CX with the sum appearing in BX–AX.

**EXAMPLE 5–7**

```
0000                    SUM32   PROC NEAR

0000 03 C1                      ADD   AX,CX
0002 13 DA                      ADC   BX,DX
0004 C3                         RET

0005                    SUM32   ENDP
```

Suppose the same procedure is rewritten for the 80386/80486/Pentium microprocessor, but modified to add two 64-bit numbers. The changes required for this operation are the use of the extended registers to hold the data and modifications of the instructions for the 80386/80486/Pentium microprocessor. These changes are shown in Example 5–8, which adds two 64-bit numbers.

**EXAMPLE 5–8**

```
0000                    SUM64   PROC NEAR

0000 66| 03 C1                 ADD   EAX,ECX
0003 66| 13 DA                 ADC   EBX,EDX
0006 C3                         RET

0007                    SUM64   ENDP
```

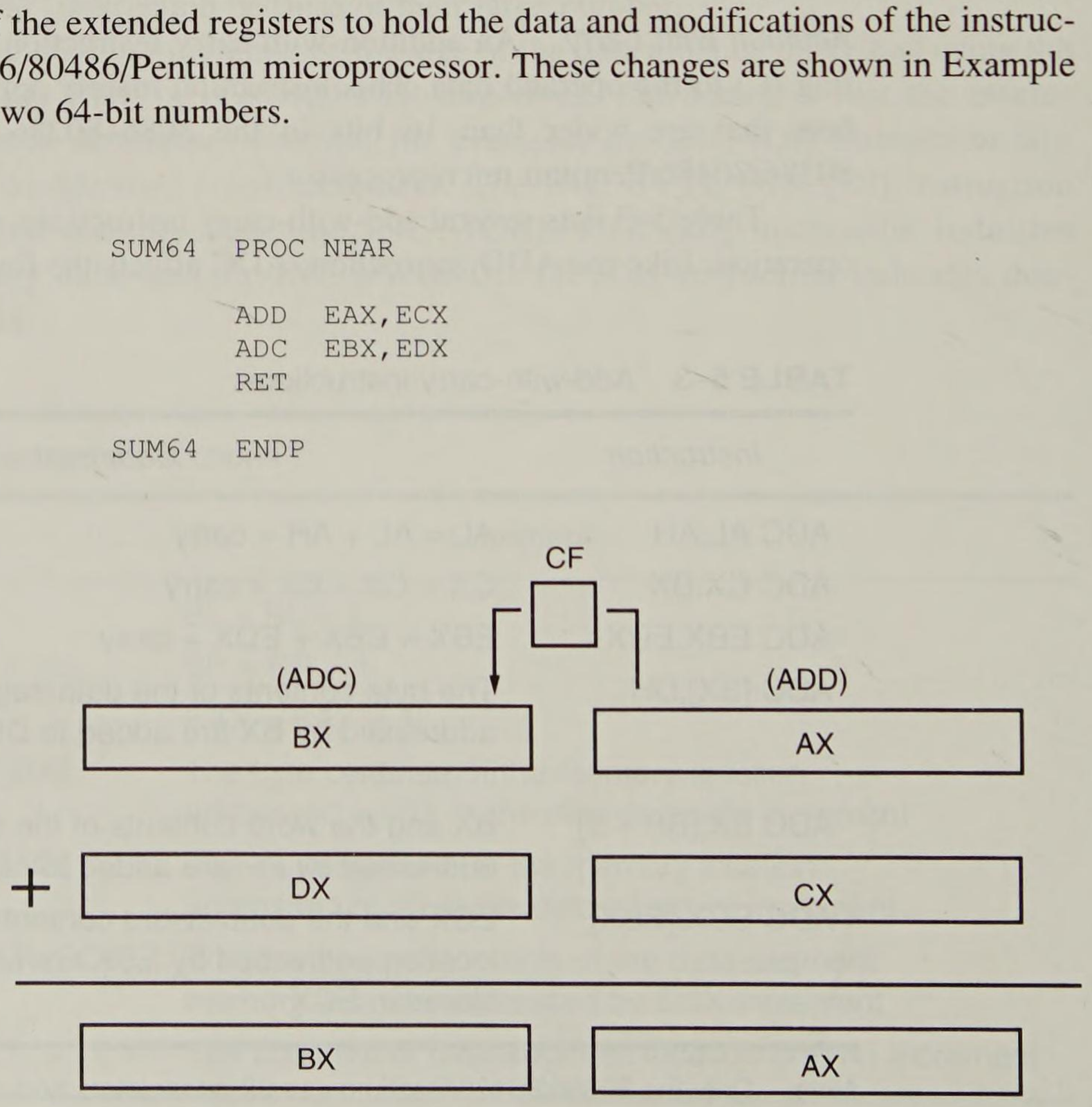



**FIGURE 5–1** Addition-with-carry showing how the carry flag (C) links the two 16-bit additions into one 32-bit addition.

*Exchange and Add for the 80486/Pentium Microprocessor.* A new type of addition, called *exchange and add* (XADD), appears in the 80486 instruction set and continues with the Pentium. The XADD instruction adds the source to the destination and stores the sum in the destination, like any addition instruction. The difference is that after the addition takes place, the original value of the destination is copied into the source operand. This is one of the few instructions that change the source.

For example, if BL = 12H, DL = 02H, and the XADD BL,DL instruction executes, the BL register contains the sum of 14H and DL becomes 12H. The sum of 14H is generated and the original destination, 12H, replaces the source. This instruction functions with any register size and any memory operand, like the ADD instruction.

## Subtraction

Many forms of subtraction (SUB) appear in the 8086–80486 and Pentium instruction sets. These forms use any addressing mode with 8-, 16-, or 32-bit data. A special form of subtraction (*decrement* or DEC) subtracts a one from any register or memory location. Section 5–3 shows how BCD and ASCII data subtract. Table 5–4 lists some of the many addressing modes allowed with the subtract instruction (SUB). There are well over 1,000 possible subtraction instructions, far too many to list. The only types of subtraction not allowed are memory-to-memory and segment register subtractions. Like other arithmetic instructions, the subtract instruction affects the flag bits.

**TABLE 5–4** Subtraction instructions

| *Instruction* | *Comment* |
|---|---|
| SUB CL,BL | CL = CL – BL |
| SUB AX,SP | AX = AX – SP |
| SUB ECX,EBP | ECX = ECX – EBP |
| SUB DH,6FH | DH = DH – 6FH |
| SUB AX,0CCCCH | AX = AX – 0CCCCH |
| SUB EAX,23456H | EAX = EAX – 00023456H |
| SUB [DI],CH | CH subtracts from the byte contents of the data-segment memory location addressed by DI |
| SUB CH,[BP] | The byte contents of the stack-segment memory location addressed by BP subtracts from CH |
| SUB AH,TEMP | The byte contents of the data-segment memory location TEMP subtracts from AH |
| SUB DI,TEMP[BX] | The word contents of the data-segment memory location addressed by TEMP + BX subtracts from DI |
| SUB ECX,DATA1 | The doubleword contents of the data-segment memory location addressed as DATA1 subtracts from ECX |

*Note*: Only the 80386/80486/Pentium use 32-bit registers and addressing modes.

Numbers that are wider than 16 bits or 32 bits must occasionally be subtracted. The subtract-with-borrow instruction (SBB), which will be discussed later, performs this type of subtraction.

**EXAMPLE 5–9**

```
0000 2B D9                      SUB   BX,CX
0002 2B DA                      SUB   BX,DX
```

*Register Subtraction.* Example 5–9 shows a sequence of instructions that perform register subtraction. This example subtracts the 16-bit contents of registers CX and DX from the contents of register BX. After each subtraction, the microprocessor modifies the contents of the flag register. The flags change for most arithmetic and logic operations.

**EXAMPLE 5–10**

```
0000 B5 22                      MOV   CH,22H
0002 80 ED 44                   SUB   CH,44H
```

*Immediate Subtraction.* The microprocessor allows immediate operands for the subtraction of constant data. Example 5–10 presents a short sequence of instructions that subtract a 44H from a 22H. Here, we first load the 22H into CH using an immediate move instruction. Next, a SUB instruction, using immediate data 44H, subtracts a 44H from the 22H. After the subtraction, the difference (DEH) moves into the CH register. The flags change as follows for this subtraction:

Z = 0 (result not zero)
C = 1 (borrow)
A = 1 (half-borrow)
S = 1 (result negative)
P = 1 (even parity)
O = 0 (no overflow)

Both carry flags (C and A) hold borrows after a subtraction, instead of carries as after an addition. In this example, there is no overflow. This example subtracted a 44H (+68) from a 22H (+34) resulting in a DEH (–34). Because the correct 8-bit signed result is a –34, there is no overflow in this example. An 8-bit overflow only occurs if the signed result is greater than +127 or less than –128.

*Decrement Subtraction.* Decrement subtraction (DEC) subtracts a one from a register or the contents of a memory location. Table 5–5 lists some decrement instructions that illustrate register and memory decrements.

The decrement-indirect memory-data instructions require BYTE PTR, WORD PTR, or DWORD PTR because the assembler cannot distinguish a byte from a word when an index register addresses memory. For example, DEC [SI] is vague, because the assembler cannot determine if the location addressed by SI is a byte or a word. Using DEC BYTE PTR [SI], DEC WORD PTR [DI], or DEC DWORD PTR [SI] reveals the size of the data.

*Subtract with Borrow.* A subtract-with-borrow (SBB) instruction functions as a regular subtraction, except that the carry flag (C), which holds the borrow, also subtracts from the

**TABLE 5–5** Decrement instructions

| *Instruction* | *Comment* |
|---|---|
| DEC BH | BH = BH – 1 |
| DEC SP | SP = SP – 1 |
| DEC ECX | ECX = ECX – 1 |
| DEC BYTE PTR [DI] | The byte contents of the data-segment memory location addressed by DI decrements |
| DEC WORD PTR [BP] | The word contents of the stack-segment memory location addressed by BP decrements |
| DEC DWORD PTR [EBX] | The doubleword contents of the data-segment memory location addressed by EBX decrements |
| DEC NUMB | The contents of data-segment memory location NUMB decrement (the way that NUMB is defined determines whether this is a byte or word decrement) |

*Note*: Only the 80386/80486/Pentium use 32-bit registers and addressing modes.

difference. The most common use for this instruction is subtractions that are wider than 16 bits in the 8086–80286 microprocessor or wider than 32 bits in the 80386/80486/Pentium. Wide subtractions require that borrows propagate through the subtraction, just as wide additions propagate the carry.

Table 5–6 lists many SBB instructions with comments that define their operation. Like the SUB instruction, SBB affects the flags. The subtract from memory immediate instruction in this table requires a BYTE PTR, WORD PTR, or DWORD PTR directive.

**TABLE 5–6** Subtract-with-borrow instructions

| *Instruction* | *Comment* |
|---|---|
| SBB | AH = AH – AL – carry |
| SBB AX,BX | AX = AX – BX – carry |
| SBB EAX,EBX | EAX = EAX – EBX – carry |
| SBB CL,3 | CL = CL – 3 – carry |
| SBB BYTE PTR [DI],3 | 3 and carry subtract from the byte contents of the data-segment memory location addressed by DI |
| SBB [DI],AL | AL and carry subtract from the byte contents of the data-segment memory location addressed by DI |
| SBB DI,[BP + 2] | The word contents of the stack-segment memory location addressed by BP, plus 2 and carry, subtract from DI |
| SBB AL,[EBX+ECX] | The byte contents of the data-segment memory location addressed by the sum of EBX and ECX, and carry, subtract from AL |

*Note*: Only the 80386/80486/Pentium use 32-bit registers and addressing modes.

When the 32-bit number held in BX and AX is subtracted from the 32-bit number held in SI and DI, the carry flag propagates the borrow between the two 16-bit subtractions required to perform this operation in the 8086–80486 and Pentium microprocessors. Figure 5–2 shows how the borrow propagates through the carry flag (C) for this task. Example 5–11 shows how this subtraction is performed by a program. With wide subtraction, the least-significant 16- or 32-bit data are subtracted with the SUB instruction. All subsequent, more significant data are subtracted using the SBB instruction. The example uses the SUB instruction to subtract DI from AX, then SBB to subtract SI from BX.

**EXAMPLE 5–11**

```
0004 2B C7                      SUB   AX,DI
0006 1B DE                      SBB   BX,SI
```

## Comparison

The comparison instruction (CMP) is a subtraction that only changes the flag bits. A comparison is useful for checking the entire contents of a register or a memory location against another value. A CMP is normally followed by a conditional jump instruction, which tests the condition of the flag bits.

Table 5–7 lists a variety of comparison instructions that use the same addressing modes as the addition and subtraction instructions already presented. The only disallowed forms of compare are memory-to-memory and segment-register compares.

**EXAMPLE 5–12**

```
0000 3C 10                      CMP   AL,10H          ;compare with 10H
0002 73 1C                      JAE   SUBER           ;if 10H or above
```

Example 5–12 shows a comparison followed by a conditional jump instruction. In this example, the contents of AL are compared with a 10H. Conditional jump instructions that often follow the compare are: JA (jump above) or JB (jump below). If the JA instruction follows the compare, the jump occurs if the value in AL is above 10H. If the JB instruction follows the compare, the jump occurs if the value in AL is below 10H. In this

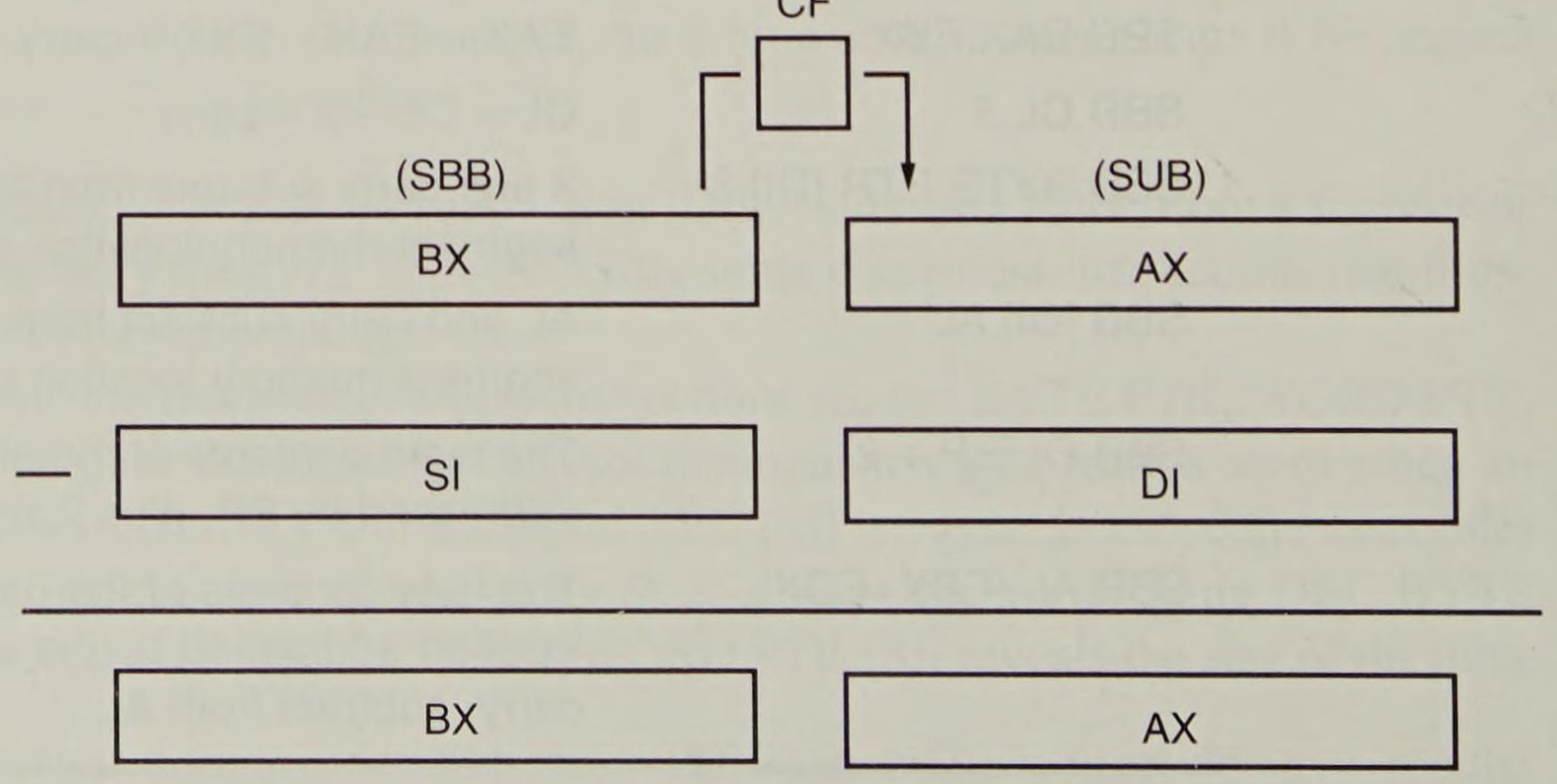



**FIGURE 5–2** Subtraction-with-borrow showing how the carry flag (C) propagates the borrow.

**TABLE 5–7** Comparison instructions

| *Instruction* | *Comment* |
|---|---|
| CMP CL,BL | Subtracts BL from CL; neither BL nor CL changes, but the flags change |
| CMP AX,SP | Subtracts SP from AX; neither AX nor SP changes, but the flags change |
| CMP EBP,ESI | Subtracts ESI from EBP; neither EBP nor ESI changes, but the flags change |
| CMP AX,0CCCCH | Subtracts 0CCCCH from AX; AX does not change, but the flags change |
| CMP [DI], CH | Subtracts CH from the byte contents of the data-segment memory location addressed by DI; neither CH nor memory changes, but the flags change |
| CMP CL,[BP] | Subtracts the byte contents of the stack-segment memory location addressed by BP from CL; neither CL nor memory changes, but the flags change |
| CMP AH,TEMP | Subtracts the byte contents of the data-segment memory location addressed by TEMP from AH; neither AH nor memory changes, but the flags change |
| CMP DI,TEMP [BX] | Subtracts the word contents of the data-segment memory location addressed by TEMP plus BX from DI; neither DI nor memory changes, but the flags change |
| CMP AL,[EDI+ESI] | Subtracts the byte contents of the data-segment memory location addressed by the sum of EDI and ESI; neither AL nor memory changes, but the flags change |

*Note*: Only the 80386/80486/Pentium use 32-bit registers and addressing modes.

example, the JAE instruction follows the compare. This instruction causes the program to continue at memory location SUBER if the value in AL is 10H or above. There is also a JBE (jump below or equal) instruction that could follow the compare to jump if the outcome is below or equal to 10H. Chapter 6 provides more details on the compare instruction and conditional jump instructions.

*Compare and Exchange (80486/Pentium Only).* The compare-and-exchange instruction (CMPXCHG), found only in the 80486 and Pentium instruction sets, compares the destination operand with the accumulator. If they are equal, the source operand is copied into the destination. If they are not equal, the destination operand is copied into the accumulator. This instruction functions with 8-, 16-, or 32-bit data.

The CMPXCHG CX,DX instruction is an example of the compare-and-exchange instruction. This instruction first compares the contents of CX with AX. If CX equals AX, DX is copied into AX. If CX is not equal to AX, CX is copied into AX. This instruction also compares AL with 8-bit data and EAX with 32-bit data, if the operands are either 8- or 32-bit.

## 5–2 MULTIPLICATION AND DIVISION

Only more modern microprocessors contain multiplication and division instructions. Early 8-bit microprocessors could not multiply or divide without the use of a program that multiplied or divided using a series of shifts and additions or subtractions. Because microprocessor manufacturers were aware of this inadequacy, they incorporated multiplication and division instructions into the instruction sets of the newer microprocessors. In fact, the Pentium contains special circuitry that performs in one clocking period a multiplication that took more than 40 clocking periods in earlier Intel microprocessors.

### Multiplication

Multiplication is performed on bytes, words, or doublewords and can be signed integer (IMUL) or unsigned integer (MUL). Only the 80386/80486/Pentium multiplies 32-bit doublewords. The product after a multiplication is always a double-width product. If we multiply two 8-bit numbers, they generate a 16-bit product; if we multiply two 16-bit numbers, they generate a 32-bit product; and if we multiply two 32-bit numbers, a 64-bit product is generated.

Some flag bits (O and C) change when the multiply instruction executes, producing predictable outcomes. Other flags also change, but their results are unpredictable, and therefore they are unused. In an 8-bit multiplication, if the most-significant 8 bits of the result are 0, both C and O flag bits equal 0. These flag bits show that the result is 8 bits wide (C = 0) or 16 bits wide (C = 1). In a 16-bit multiplication, if the most-significant 16 bits of the product are 0, both C and O clear to 0. In a 32-bit multiplication, both C and O indicate that the most-significant 32 bits of the product are zero.

*8-Bit Multiplication.* With 8-bit multiplication, whether signed or unsigned, the multiplicand is always in the AL register. The multiplier can be any 8-bit register or any memory location. Immediate multiplication is not allowed unless the signed immediate multiplication instruction, discussed later in this section, appears in a program. The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL. An example is the MUL BL instruction, which multiplies the unsigned contents of AL by the unsigned contents of BL. After the multiplication, the unsigned product is placed in AX—a double-width product. Table 5–8 illustrates some of 8-bit multiplication instructions.

Suppose that BL and CL each contain two 8-bit unsigned numbers, and these numbers must be multiplied to form a 16-bit product stored in DX. This procedure cannot be accomplished by a single instruction because we can only multiply a number times the AL register for an 8-bit multiplication. Example 5–13 shows a short program that generates DX = BL × CL. This example loads register BL and CL with example data 5 and 10. The product, a 50, moves into DX from AX after the multiplication by using the MOV DX,AX instruction.

**EXAMPLE 5–13**

```
0000 B3 05                      MOV   BL,5              ;load data
0002 B1 0A                      MOV   CL,10
```

```
0004 8A C1                  MOV  AL,CL          ;position data
0006 F6 E3                  MUL  BL             ;multiply
0008 8B D0                  MOV  DX,AX          ;position product
```

For signed multiplication, the product is in true binary form if positive, and in two's complement form if negative. These are the same forms used to store all positive and negative signed numbers used by the microprocessor. If the program in Example 5–13 multiplies two signed numbers, only the MUL instruction is changed to IMUL.

*16-Bit Multiplication.* Word multiplication is very similar to byte multiplication. The difference is that AX, rather than AL, contains the multiplicand and the product appears in DX–AX instead of AX. The DX register always contains the most-significant 16 bits of the product and the AX register always contains the least-significant 16 bits of the product. The choice of the multiplier is up to the programmer. Table 5–9 shows several different 16-bit multiplication instructions.

*Immediate 16-Bit Multiplication.* The 8086/8088 microprocessor could not perform immediate multiplication, but the 80286, 80386, 80486, and Pentium can, by using a special version of the multiply instruction. Immediate multiplication must be signed multiplication and the instruction format is different because it contains three operands. The first operand

**TABLE 5–8** 8-bit multiplication instructions

| *Instruction* | *Comment* |
|---|---|
| MUL CL | AL is multiplied by CL; this unsigned multiplication leaves the product in AX |
| IMUL DH | AL is multiplied by DH; this signed multiplication leaves the product in AX |
| IMUL BYTE PTR [BX] | AL is multiplied by the byte contents of the data-segment memory location addressed by BX; this unsigned multiplication leaves the product in AX |
| MUL TEMP | AL is multiplied by the contents of the data-segment memory location TEMP; if TEMP is defined as an 8-bit number, the unsigned product is found in AX |

**TABLE 5–9** 16-bit multiplication instructions

| *Instruction* | *Comment* |
|---|---|
| MUL CX | AX is multiplied by CX; the unsigned product is found in DX–AX |
| IMUL DI | AX is multiplied by DI; the signed product is found in DX–AX |
| MUL WORD PTR [SI] | AX is multiplied by the word contents of the data-segment memory location addressed by SI; the unsigned product is found in DX–AX |

is the 16-bit destination register; the second operand is a register or memory location that contains the 16-bit multiplicand; and the third operand is either 8-bit or 16-bit immediate data used as the multiplier.

The IMUL CX,DX,12H instruction multiplies 12H times DX, and leaves a 16-bit signed product in CX. If the immediate data are 8-bit they sign-extend into a 16-bit number before the multiplication occurs. Another example is IMUL BX,NUMBER,1000H, which multiplies NUMBER times 1000H, and leaves the product in BX. Both the destination and multiplicand must be 16-bit numbers. Although this is immediate multiplication, its utility is limited by the fact that it is a signed multiplication and the product is 16 bits wide.

*32-Bit Multiplication.* In the 80386, 80486, and Pentium microprocessors, 32-bit multiplication is allowed because these microprocessors contain 32-bit registers. With 32-bit multiplication, we can multiply signed or unsigned numbers using the same IMUL and MUL instructions used with 8- and 16-bit multiplication. With 32-bit multiplication, the contents of EAX are multiplied by the operand specified with the instruction. The product (64 bits wide) is found in EDX–EAX, where EAX contains the least-significant 32 bits of the product. Table 5–10 lists some of the 32-bit multiplication instructions found in the 80386/80486/Pentium instruction set.

## Division

Like multiplication, division occurs on 8- or 16-bit numbers in the 8086–80486/Pentium and on 32-bit numbers in the 80386/80486/Pentium. These numbers are signed (IDIV) or unsigned (DIV) integers. The dividend is always a double-width dividend divided by the operand. This means an 8-bit division divides a 16-bit number by an 8-bit number, a 16-bit division divides a 32-bit number by a 16-bit number, and a 32-bit division divides a 64-bit number by a 32-bit number. There is no immediate division instruction available to any 8086–80486 and Pentium microprocessors.

None of the flag bits change predictably for a division. Two different kinds of division can result in errors. One of these is an attempt to divide by zero and the other is a divide overflow. A divide overflow occurs when a small number divides into a large number. For example, suppose that AX = 3,000 and that we divide it by 2. Because the quotient for an 8-bit division appears in AL, the result, 1,500, does not fit into AL, causing a divide overflow. The microprocessor generates an interrupt if any kind of divide error occurs. In most cases, a divide error interrupt displays an error message on the video

**TABLE 5–10** 32-bit multiplication instructions

| *Instruction* | *Comment* |
|---|---|
| MUL ECX | EAX is multiplied by ECX; the unsigned product is found in EDX–EAX |
| IMUL EDI | EAX is multiplied by EDI; the signed product is found in EDX–EAX |
| MUL DWORD PTR[ECX] | EAX is multiplied by the doubleword contents of the data-segment memory location addressed by ECX; the unsigned product is found in EDX–EAX |

screen. The divide error interrupt, and all other interrupts for the microprocessor, are explained in Chapter 6.

*8-Bit Division.* An 8-bit division uses the AX register to store the dividend that is divided by the contents of any 8-bit register or memory location. The quotient moves into AL after the division, with AH containing a whole number remainder. For a signed division, the quotient is positive or negative and the remainder *always* assumes the sign of the dividend and is always an integer. For example, if AX = 0010H (+16) and BL = FDH (–3) and the IDIV BL instruction executes, AX = 01FBH. This represents a quotient of –5 (AL) with a remainder of 1 (AH). On the other hand, if a –16 is divided by a +3, the result will be a quotient of –5 (AL) with a remainder of –1 (AH). Table 5–11 lists some of the 8-bit division instructions.

With 8-bit division, the numbers are usually 8 bits wide. This means that one of them, the dividend, must be converted to a 16-bit-wide number in AX. This is accomplished differently for signed and unsigned numbers. For an unsigned number, the most-significant 8 bits must be cleared to zero *(zero-extended)*. The MOVZX instruction described in Chapter 4 can be used to zero-extend a number in the 80386/80486/Pentium microprocessor. For a signed number, the least-significant 8 bits are sign-extended into the most-significant 8 bits. A special instruction exists that sign-extends AL into AH, or converts an 8-bit signed number in AL into a 16-bit signed number in AX. The CBW *(convert byte to word)* instruction performs this conversion. In the 80386/80486/Pentium microprocessor a MOVSX instruction (see Chapter 4) can sign-extend a number.

**EXAMPLE 5–14**

```
0000 A0 0000 R          MOV   AL,NUMB          ;get NUMB
0003 B4 00              MOV   AH,0             ;zero-extend
0005 F6 36 0002 R       DIV   NUMB1            ;divide by NUMB1
0009 A2 0003 R          MOV   ANSQ,AL          ;save quotient
000C 88 26 0004 R       MOV   ANSR,AH          ;save remainder
```

Example 5–14 illustrates a short program that divides the unsigned byte contents of memory location NUMB by the unsigned contents of memory location NUMB1. Here we store the quotient in location ANSQ and the remainder in location ANSR. Notice how the contents of location NUMB are retrieved from memory and then zero-extended to form a 16-bit unsigned number for the dividend.

**TABLE 5–11** 8-bit division instructions

| *Instruction* | *Comment* |
|---|---|
| DIV CL | AX is divided by CL; the unsigned quotient is in AL and the remainder is in AH |
| IDIV BL | AX is divided by BL; the signed quotient is in AL and the remainder is in AH |
| DIV BYTE PTR [BP] | AX is divided by the byte contents of the stack-segment memory location addressed by BP; the unsigned quotient is in AL and the remainder is in AH |

**EXAMPLE 5–15**

```
0000 A0 0000 R          MOV  AL,NUMB        ;get NUMB
0003 98                 CBW                 ;sign-extend
0004 F6 3E 0002 R       IDIV NUMB1          ;divide by NUMB1
0008 A2 0003 R          MOV  ANSQ,AL        ;save quotient
000B 88 26 0004 R       MOV  ANSR,AH        ;save remainder
```

Example 5–15 shows the same basic program, except that the numbers are signed numbers. This means that instead of zero-extending AL into AH, we sign-extend it using the CBW instruction.

*16-Bit Division.* Sixteen-bit division is similar to 8-bit division, except that instead of dividing into AX we divide into DX–AX, a 32-bit dividend. The quotient appears in AX and the remainder in DX after a 16-bit division. Table 5–12 lists some of the 16-bit division instructions.

As with 8-bit division, numbers must often be converted to the proper form for the dividend. If we start with a 16-bit unsigned number in AX, then DX must be cleared to zero. In the 80386/80486/Pentium, the number is zero-extended using the MOVZX instruction, If AX is a 16-bit signed number, the CWD (*convert word to doubleword*) instruction sign-extends it into a signed 32-bit number. If an 80386/80486/Pentium microprocessor is available, the MOVSX instruction can also be used to sign-extend a number.

**EXAMPLE 5–16**

```
0000 B8 FF9C            MOV  AX,-100        ;load -100
0003 B9 0009            MOV  CX,9           ;load +9
0006 99                 CWD                 ;sign-extend
0007 F7 F9              IDIV CX
```

Example 5–16 shows the division of two 16-bit signed numbers. Here a –100 in AX is divided by a +9 in CX. The CWD instruction converts the –100 in AX to a –100 in DX–AX before the division. After the division, the results appear in DX–AX as a quotient of –11 in AX and a remainder of –1 in DX.

*32-Bit Division.* The 80386/80486/Pentium microprocessor performs 32-bit division on signed or unsigned numbers. The 64-bit contents of EDX–EAX are divided by the operand

**TABLE 5–12** 16-bit division instructions

| *Instruction* | *Comment* |
|---|---|
| DIV CX | DX–AX is divided by CX; the unsigned quotient is in AX and the remainder is in DX |
| IDIV SI | DX–AX is divided by SI; the signed quotient is in AX and the remainder is in DX |
| DIV NUMB | DX–AX is divided by the word contents of the data-segment memory location NUMB; the unsigned quotient is in AX and the remainder is in DX |

**TABLE 5–13** 32-bit division instructions

| *Instruction* | *Comment* |
|---|---|
| DIV ECX | EDX–EAX is divided by ECX; the unsigned quotient is in EAX and the remainder is in EDX |
| DIV DATA2 | EDX–EAX is divided by the doubleword contents of the data-segment memory location addressed by DATA2; the unsigned quotient is in EAX and the remainder is in EDX |
| IDIV DWORD PTR [EDI] | EDX–EAX is divided by the doubleword contents of the data-segment memory location addressed by EDI; the signed quotient is in EAX and the remainder is in EDX |

specified by the instruction leaving a 32-bit quotient in EAX and a 32-bit remainder in EDX. Other than the size of the registers, this instruction functions in the same manner as the 8- and 16-bit divisions. Table 5–13 shows some example 32-bit division instructions. The convert doubleword to quadword instruction (CDQ) is used before a signed division to convert the 32-bit contents of EAX into a 64-bit signed number in EDX–EAX.

*The Remainder.* What do we do with the remainder after a division? There are a few possible choices. We could use the remainder to round the result, or we could drop the remainder to truncate the result. If the division is unsigned, rounding requires that the remainder is compared with half the divisor to decide whether to round the quotient up or down. We could also convert the remainder to a fraction.

**EXAMPLE 5–17**

```
0000 F6 F3                      DIV  BL              ;divide
0002 02 E4                      ADD  AH,AH           ;double remainder
0004 3A E3                      CMP  AH,BL           ;test for rounding
0006 72 02                      JB   NEXT
0008 FE C0                      INC  AL              ;round
000A                       NEXT:
```

Example 5–17 shows a sequence of instructions that divide AX by BL and round the result. This program doubles the remainder before comparing it with BL, to decide whether to round the quotient. Here, an INC instruction rounds the contents of AL after the compare.

Suppose we need a fractional remainder instead of an integer remainder. A fractional remainder is obtained by saving the quotient. Next, the AL register is cleared to zero. The number remaining in AX is now divided by the original operand to generate a fractional remainder.

**EXAMPLE 5–18**

```
0000 B8 000D                    MOV  AX,13           ;load 13
0003 B3 02                      MOV  BL,2            ;load 2
0005 F6 F3                      DIV  BL              ;13/2
0007 A2 0003 R                  MOV  ANSQ,AL         ;save quotient
```

```
000A B0 00                  MOV  AL,0           ;clear AL
000C F6 F3                  DIV  BL             ;generate remainder
000E A2 0004 R              MOV  ANSR,AL        ;save remainder
```

Example 5–18 shows how a 13 is divided by a 2. The 8-bit quotient is saved in memory location ANSQ and then AL is cleared. Next the contents of AX are again divided by 2, to generate a fractional remainder. After the division, the AL register equals an 80H. This is a $10000000_2$. If the binary point (radix) is placed before the left-most bit of AL, we have a fractional remainder of $0.10000000_2$, or 0.5 decimal. The remainder is saved in memory location ANSR in this example.

## 5–3 BCD AND ASCII ARITHMETIC

The 8086–80486 and Pentium microprocessors allow arithmetic manipulation of both binary-coded decimal (BCD) and American Standard Code for Information Interchange (ASCII) data. This is accomplished by instructions that adjust the numbers for BCD and ASCII arithmetic.

The BCD operations occur in systems such as point-of-sales terminals (cash registers) and others that seldom require arithmetic. The ASCII operations are performed on ASCII data used by many programs. Today, we rarely use BCD or ASCII arithmetic.

### BCD Arithmetic

Two arithmetic techniques operate with BCD data: addition and subtraction. The instruction set provides two instructions that correct the result of a BCD addition and a BCD subtraction. The DAA (*decimal adjust after addition*) instruction follows BCD addition and the DAS (*decimal adjust after subtraction*) instruction follows BCD subtraction. Both instructions correct the result of the addition or subtraction so it is a BCD number.

For BCD data, the numbers always appear in the packed BCD form and are stored as two BCD digits per byte. The adjust instructions only function with the AL register after BCD addition and subtraction.

*DAA Instruction.* The DAA instruction follows the ADD or ADC instruction to adjust the result into a BCD result. Suppose that DX and BX each contain 4-digit packed BCD numbers. Example 5–19 provides a short sample program that adds the BCD numbers in DX and BX, and stores the result in CX.

**EXAMPLE 5–19**

```
0000 BA 1234                MOV  DX,1234H       ;load 1,234
0003 BB 3099                MOV  BX,3099H       ;load 3,099
0006 8A C3                  MOV  AL,BL          ;sum BL with DL
0008 02 C2                  ADD  AL,DL
000A 27                     DAA                 ;adjust
000B 8A C8                  MOV  CL,AL          ;answer to CL
000D 8A C7                  MOV  AL,BH          ;sum BH, DH, and CF
000F 12 C6                  ADC  AL,DH
0011 27                     DAA                 ;adjust
0012 8A E8                  MOV  CH,AL          ;answer to CH
```

Because the DAA instruction only functions with the AL register, this addition must occur 8 bits at a time. After adding the BL and DL registers, the result is adjusted with a DAA instruction before being stored in CL. Next, we add BH and DH registers with carry, and the result again is adjusted with DAA before being stored in CH. In this example, a 1,234 adds to a 3,099 to generate a sum of 4,333 that moves into CX after the addition. Note that 1234 BCD is the same as 1234H.

**EXAMPLE 5–20**

```
0000 BA 1234          MOV  DX,1234H        ;load 1,234
0003 BB 3099          MOV  BX,3099H        ;load 3,099
0006 8A C3            MOV  AL,BL           ;subtract DL from BL
0008 2A C2            SUB  AL,DL
000A 2F               DAS                  ;adjust
000B 8A C8            MOV  CL,AL           ;answer to CL
000D 8A C7            MOV  AL,BH           ;subtract DH
000F 1A C6            SBB  AL,DH
0011 2F               DAS                  ;adjust
0012 8A E8            MOV  CH,AL           ;answer to CH
```

*DAS instruction.* The DAS instruction functions just like the DAA instruction, except it follows a subtraction instead of an addition. Example 5–20 is basically the same as Example 5–19, except that it subtracts instead of adds DX and BX. The main difference in these programs is that the DAA instruction changes to DAS, and the ADD and ADC instructions change to SUB and SBB.

## ASCII Arithmetic

The ASCII arithmetic instructions function with ASCII-coded numbers. These numbers range in value from 30H to 39H for the numbers 0–9. There are four instructions used with ASCII arithmetic operations: AAA (*ASCII adjust after addition*), AAD (*ASCII adjust before division*), AAM (*ASCII adjust after multiplication*), and AAS (*ASCII adjust after subtraction*). These instructions use register AX as the source and as the destination.

*AAA Instruction.* The addition of two 1-digit ASCII-coded numbers will not result in any useful data. For example, if we add a 31H and 39H, the result is a 6AH. This ASCII addition (1 + 9) should produce a 2-digit ASCII result equivalent to a 10 decimal, which is a 31H and a 30H in ASCII code. If we execute the AAA instruction after this addition, the AX register will contain a 0100H. Although this is not ASCII code, it can be converted to ASCII code by adding 3030H, which generates a 3130H. The AAA instruction *clears* AH if the result is less than 10, and *adds* a 1 to AH if the result is greater than 10.

**EXAMPLE 5–21**

```
0000 B8 0031          MOV  AX,31H          ;load ASCII 1
0003 04 39            ADD  AL,39H          ;add ASCII 9
0005 37               AAA                  ;adjust
0006 05 3030          ADD  AX,3030H        ;answer to ASCII
```

Example 5–21 shows how ASCII addition functions in the microprocessor. Please note that we clear AH before the addition by using the MOV AX,31H instruction. The operand of 0031H places a 00H in AH and a 31H into AL.

*AAD Instruction.* Unlike all the other adjust instructions, the AAD instruction appears before a division. The AAD instruction requires that the AX register contains a 2-digit unpacked BCD (not ASCII) number before executing. After adjusting the AX register with AAD, it is divided by an unpacked BCD number to generate a single-digit result in AL, with any remainder in AH.

**EXAMPLE 5–22**

```
0000 B3 09          MOV  BL,9          ;load divisor
0002 B8 0702        MOV  AX,0702H      ;load dividend
0005 D5 0A          AAD                ;adjust
0007 F6 F3          DIV  BL
```

Example 5–22 illustrates how a 72 in unpacked BCD is divided by 9 to produce a quotient of 8. The 0702H loaded into the AX register is adjusted by the AAD instruction to 0048H. This converts a 2-digit unpacked BCD number into a binary number so it can be divided with the binary division instruction (DIV). The AAD instruction converts the unpacked BCD numbers between 00 and 99 into binary.

**EXAMPLE 5–23**

```
0000 B0 05          MOV  AL,5          ;load multiplicand
0002 B1 05          MOV  CL,5          ;load multiplier
0004 F6 E1          MUL  CL
0006 D4 0A          AAM                ;adjust
```

*AAM Instruction.* The AAM instruction follows the multiplication instruction after multiplying two 1-digit unpacked BCD numbers. Example 5–23 shows a short program that multiplies a 5 times a 5. The result after the multiplication is 0018H in the AX register. After adjusting the result with the AAM instruction, AX contains a 0205H. This is an unpacked BCD result of 25. If 3030H adds to 0205H, this becomes an ASCII result of 3235H.

The AAM instruction accomplishes this conversion by dividing AX by 10. The remainder is found in AL and the quotient is in AH. The second byte of the instruction contains a 0AH. If the 0AH is changed to another value, AAM divides by the new value. For example, if the second byte is changed to a 0BH, the AAM instruction divides by an 11.

One benefit of the AAM instruction is that AAM converts from binary to unpacked BCD. If a binary number between 0000H and 0063H appears in the AX register, the AAM instruction converts it to BCD. For example, if AX contains a 0060H before AAM, it will contain a 0906H after AAM executes. This is the unpacked BCD equivalent of 96 decimal. If 3030H is added to 0906H, we have changed the result to ASCII.

**EXAMPLE 5–24**

```
0000 33 D2          XOR  DX,DX         ;clear DX register
0002 B9 0064        MOV  CX,100        ;divide DX—AX by 100
0005 F7 F1          DIV  CX            ;AX=quotient and DX=remainder
0007 D4 0A          AAM                ;convert quotient to BCD
0009 05 3030        ADD  AX,3030H      ;convert to ASCII
000C 92             XCHG AX,DX         ;repeat for remainder
000D D4 0A          AAM
000F 05 3030        ADD  AX,3030H
```

Example 5–24 shows how the 16-bit binary contents of AX are converted to a 4-digit ASCII character string by using division and the AAM instruction. This works for numbers between 0 and 9,999. First DX is cleared and then DX–AX is divided by a 100. For example, if AX = 245 after the division, AX = 2 and DX = 45. These separate halves are converted to BCD using AAM, and then a 3030H is added to convert to ASCII code.

*AAS Instruction.* Like other ASCII adjust instructions, AAS adjusts the AX register after an ASCII subtraction. For example, suppose a 35H subtracts from a 39H. The result will be a 04H, which requires no correction. Here AAS will modify neither AH nor AL. But if 38H subtracts from 37H, then AL will equal 09H and the number in AH will decrement by 1. This decrement allows multiple-digit ASCII numbers to be subtracted from each other.

## 5–4 BASIC LOGIC INSTRUCTIONS

The basic logic instructions include: AND, OR, Exclusive-OR, and NOT. Another logic instruction, TEST, is explained in this section of the text because it is a special form of the AND instruction. Also explained is the NEG instruction, which is similar to the NOT instruction.

Logic operations provide binary bit control in *low-level software*. The logic instructions allow bits to be set, cleared, or complemented. Low-level software appears in machine language or assembly language form and often controls the I/O devices in a system. All logic instructions affect the flag bits. Logic operations always clear the carry and overflow flags, while the other flags change to reflect the condition of the result.

When binary data are manipulated in a register or a memory location, the right-most bit position is always numbered bit 0. Bit position numbers increase from bit 0, toward the left to bit 7 for a byte and to bit 15 for a word. A doubleword (32 bits) uses bit position 31 as its left-most bit.

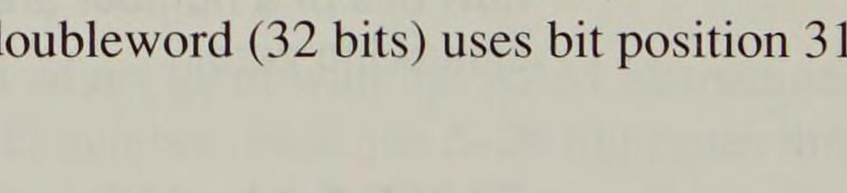
### AND

The AND operation performs logical multiplication as illustrated by the truth table in Figure 5–3. Here, two bits, A and B, are ANDed to produce the result X. As indicated by the truth table, X is a logic 1 only when both A and B are logic 1s. For all other input combination of A and B, X is a logic 0. It is important to remember that 0 AND anything is always 0, and 1 AND 1 is always a 1.

**FIGURE 5–3** (a) The truth table for the AND operation and (b) the logic symbol of an AND gate.

| A | B | T |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a)




(b)

The AND instruction can replace discrete AND gates if the speed required is not too great. With the 8086–80486 and Pentium microprocessors, the AND instruction often executes in about a microsecond. So if the circuit the AND instruction replaces operates at a slower speed than a microsecond, the AND instruction is a logical replacement. This replacement saves a considerable amount of money. A single AND gate integrated circuit (7408) costs approximately 40¢, while it costs less than 1/100¢ to store the AND instruction in a read-only memory.

The AND operation also clears bits in a binary number. The task of clearing a bit in a binary number is called *masking*. Figure 5–4 illustrates the process of masking. Notice that the left-most four bits clear to 0, because 0 AND anything is 0. The bit positions that AND with 1s do not change. This occurs because if a 1 ANDs with a 1, a 1 results, and if a 1 ANDs with a 0, a 0 results.

The AND instruction uses any addressing mode except memory-to-memory and segment-register. Refer to Table 5–14 for a list of some AND instructions and a comment about their operation.

**EXAMPLE 5–25**

```
0000 BB 3135                MOV  BX,3135H        ;load ASCII
0003 81 E3 0F0F             AND  BX,0F0FH        ;mask BX
```

An ASCII-coded number can be converted to BCD by using the AND instruction to mask off the left-most four binary bit positions. This converts the ASCII 30H to 39H to 0–9.

```
    x x x x  x x x x   Unknown number
•   0 0 0 0  1 1 1 1   Mask
-------------------
    0 0 0 0  x x x x   Result
```

**FIGURE 5–4** The operation of the AND function showing how bits of a number are cleared to zero.

**TABLE 5–14** AND instructions

| *Instruction* | *Comment* |
|---|---|
| AND AL,BL | AL = AL AND BL |
| AND CX,DX | CX = CX AND DX |
| AND ECX,EDI | ECX = ECX AND EDI |
| AND CL,33H | CL = CL AND 33H |
| AND DI,4FFFH | DI = DI AND 4FFFH |
| AND ESI,34H | ESI = ESI AND 00000034H |
| AND AX, [DI] | AX is ANDed with the word contents of the data-segment memory location addressed by DI; the result moves into AX |
| AND ARRAY [SI],AL | The byte contents of the data-segment memory location addressed by ARRAY plus SI are ANDed to AL; the result moves to memory |
| AND [EAX],CL | The byte contents of CL are ANDed with the byte contents of the data-segment memory location addressed by register EAX; the result moves to memory |

Example 5–25 shows a short program that converts the ASCII contents of BX into BCD. The AND instruction in this example converts two digits from ASCII to BCD simultaneously.

## OR

The OR operation performs logical addition and is often called the *Inclusive-OR function.* The OR function generates a logic 1 output if any inputs are 1. A 0 appears at the output only when all inputs are 0. The truth table for the OR function appears in Figure 5–5. Here, the inputs, A and B, OR together to produce the X output. Remember that 1 ORed with anything yields a 1.

The OR instruction can also replace discrete OR gates. This results in a considerable savings, because a quad, 2-input OR gate (7432) costs about 40¢, while the OR instruction costs less than $\frac{1}{100}$¢ to store in a read-only memory.

Figure 5–6 shows how the OR gate sets (1) any bit of a binary number. Here an unknown number (XXXX XXXX) ORs with a 0000 1111 to produce a result of XXXX 1111. The right-most four bits set, while the left-most four bits remain unchanged. The OR operation sets any bit and the AND operation clears any bit.

The OR instruction uses any of the addressing modes allowed to any other instruction except segment-register addressing. Table 5–15 illustrates several example OR instructions with comments about their operation.

**EXAMPLE 5–26**

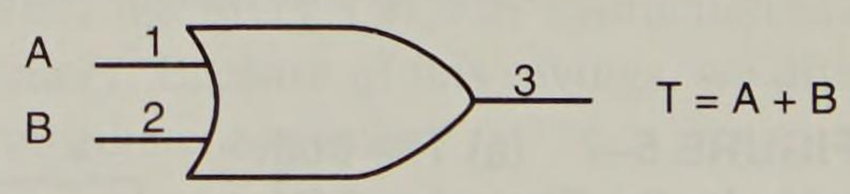
```
0000 B0 05          MOV   AL,5          ;load data
0002 B3 07          MOV   BL,7
0004 F6 E3          MUL   BL
0006 D4 0A          AAM                 ;adjust
0008 0D 3030        OR    AX,3030H      ;to ASCII
```

Suppose we multiply two BCD numbers and adjust them with the AAM instruction. The result appears in AX as a 2-digit unpacked BCD number. Example 5–26 illustrates this multiplication and shows how to change the result into a 2-digit ASCII-coded number using

**FIGURE 5–5** (a) The truth table for the OR operation and (b) the logic symbol of an OR gate.

| A | B | T |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(a)




(b)

**FIGURE 5–6** The operation of the OR function showing how bits of a number are set to one.

```
  x x x x  x x x x   Unknown number
+ 0 0 0 0  1 1 1 1   Mask
-------------------
  x x x x  1 1 1 1   Result
```

**TABLE 5–15** OR instructions

| *Instruction* | *Comment* |
|---|---|
| OR AH,BL | AH = AH OR BL |
| OR SI,DX | SI = SI OR DX |
| OR EAX,EBX | EAX = EAX OR EBX |
| OR DH,0A3H | DH = DH OR 0A3H |
| OR SP,990DH | SP = SP OR 990DH |
| OR EBP,10 | EBP = EBP OR 0000000AH |
| OR DX,[BX] | DX is ORed with the word contents of the data-segment memory location addressed by BX |
| OR DATES [DI + 2],AL | The data-segment memory location addressed by DATES plus DI plus 2 is ORed with AL |

the OR instruction. Here, OR AX,3030H converts the 0305H found in AX to 3335H. The OR operation can be replaced with an ADD AX,3030H instruction to obtain the same results.

## Exclusive-OR

The Exclusive-OR instruction (XOR) differs from Inclusive-OR (OR). The difference is that a 1,1 condition of the OR function produces a 1, while a 1,1 condition of the Exclusive-OR operation produces a 0. The Exclusive-OR operation excludes this condition, while the Inclusive-OR includes it.

Figure 5–7 shows the truth table of the Exclusive-OR function. (Compare this with Figure 5–5 to appreciate the difference between these two OR functions.) If the inputs of the Exclusive-OR function are both 0 or both 1, the output is 0. If the inputs are different, the output is a 1. Because of this, the Exclusive-OR is sometimes called a *comparator*.

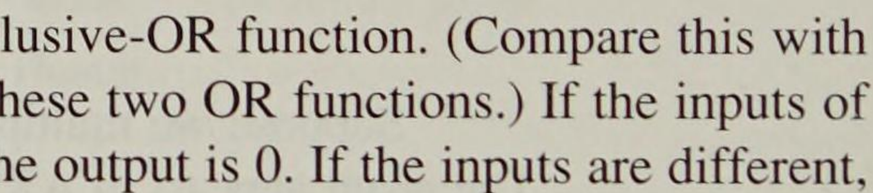
The XOR instruction uses any addressing mode except segment-register addressing. Table 5–16 lists various forms of the Exclusive-OR instruction with comments about their operation.

As with the AND and OR functions, Exclusive-OR can replace discrete logic circuitry. The 7486 quad, 2-input Exclusive-OR gate is replaced by one XOR instruction. The 7486

**FIGURE 5–7** (a) The truth table for the Exclusive-OR operation and (b) the logic symbol of an Exclusive-OR gate.

| A | B | T |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(a)




(b)

**TABLE 5–16** Exclusive-OR instructions

| *Instruction* | *Comment* |
|---|---|
| XOR CH,DL | CH = CH XOR DL |
| XOR SI,BX | SI = SI XOR BX |
| XOR EBX,EDI | EBX = EBX XOR EDI |
| XOR CH,0EEH | CH = CH XOR 0EEH |
| XOR DI,00DDH | DI = DI XOR 00DDH |
| XOR ESI,100 | ESI = ESI XOR 00000064H |
| XOR DX,[SI] | DX is XORed with the word contents of the data-segment memory location addressed by SI; the result is left in DX |
| XOR DATES[DI + 2],AL | The byte contents of the data-segment memory location addressed by DATES plus DI plus 2 are XORed with AL; the result is left in memory |

costs about 40¢, while the instruction costs less than $\frac{1}{100}$¢ to store in the memory. Replacing just one 7486 saves a considerable amount of money, especially if we build many systems.

The Exclusive-OR instruction is useful if some bits of a register or memory location must be inverted. This instruction allows part of a number to be inverted or complemented. Figure 5–8 shows how just part of an unknown quantity can be inverted by XOR. Notice that when a 1 Exclusive-ORs with X, the result is $\overline{X}$. If a 0 Exclusive-ORs with X, the result is X.

Suppose that the left-most ten bits of the BX register must be inverted without changing the right-most six bits. The XOR BX,0FFC0H instruction accomplishes this task. The AND instruction clears (0) bits, the OR instruction sets (1) bits, and now the Exclusive-OR instruction inverts bits. These three instructions allow a program to gain complete control over any bit stored in any register or memory location. This is ideal for control system applications where equipment must be turned on (1), turned off (0), and toggled from on to off or off to on.

A fairly common use for the Exclusive-OR instruction is to clear a register to zero. For example, the XOR CH,CH instruction clears register CH to 00H and requires two bytes of memory to store the instruction. Likewise, the MOV CH,00H instruction also clears CH to 00H, but requires three bytes of memory. Because of this savings, we often use XOR to clear a register rather than a move immediate instrucion.

## Test and Bit Test Instructions

The TEST instruction performs the AND operation. The AND instruction changes the destination operand, while the TEST instruction does not. TEST only affects the condition of

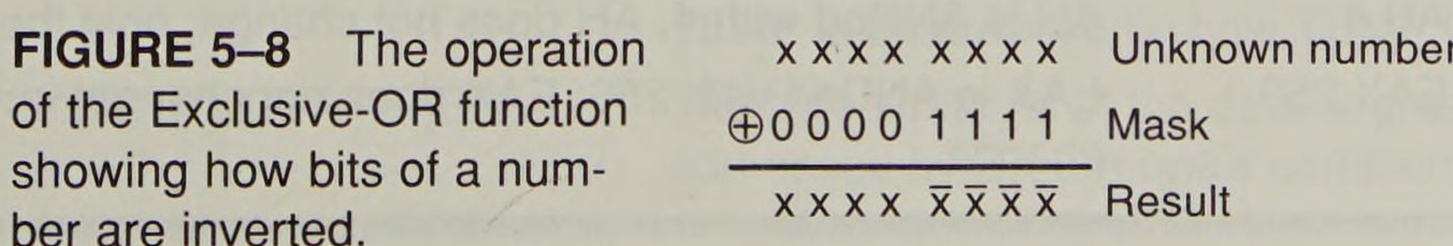



**FIGURE 5–8** The operation of the Exclusive-OR function showing how bits of a number are inverted.

the flag register, which indicates the result of the test. The TEST instruction uses the same addressing modes as the AND instruction. Table 5–17 illustrates some forms of the TEST instruction with comments about the operation of each form.

The TEST instruction functions in the same manner as a CMP instruction, but the TEST instruction normally tests a single bit (or occasionally multiple bits), while the CMP instruction tests the entire byte or word. The zero flag (Z) is a logic 1 (indicating a zero result) if the bit under test is a zero, and Z = 0 (indicating a nonzero result) if the bit under test is not a zero.

Usually the TEST instruction is followed by either the JZ (*jump zero*) or JNZ (*jump not zero*) instruction. The destination operand is normally tested against immediate data. The values of immediate data are 1 to test the right-most bit position, 2 to test the next bit, 4 for the next, etc.

Example 5–27 lists a short program that tests the right-most and left-most bit positions of the AL register. Here, 1 selects the right-most bit and 128 selects the left-most bit. The JNZ instruction follows each test to jump to different memory locations depending on the outcome of the tests. The JNZ instruction jumps to the operand address (RIGHT or LEFT in the example) if the bit under test is not zero.

**EXAMPLE 5–27**

```
0000 A8 01                TEST AL,1           ;test right bit
0002 75 1C                JNZ  RIGHT          ;if set
0004 A8 80                TEST AL,128         ;test left bit
0006 75 38                JNZ  LEFT           ;if set
```

The 80386/80486/Pentium microprocessor also contains additional test instructions that test bit positions. Table 5–18 lists the four bit test instructions available to these microprocessors.

All four forms of the bit test instruction test the bit position in the destination operand selected by the source operand. For example, the BT AX,4 instruction tests bit position 4 in AX. The result of the test is located in the carry flag bit. If bit position 4 is a 1, carry is set; if bit position 4 is a zero, carry is cleared.

**TABLE 5–17** TEST instructions

| *Instruction* | *Comment* |
|---|---|
| TEST DL,DH | DL is ANDed with DH; neither DL nor DH changes; only the flags change |
| TEST CX,BX | CX is ANDed with BX; neither CX nor BX changes; only the flags change |
| TEST EDX,ECX | EDX is ANDed with ECX; neither EDX nor ECX changes; only the flags change |
| TEST AH,4 | AH is ANDed with 4; AH does not change; only the flags change |
| TEST EAX,256 | EAX is ANDed with 256; EAX does not change; only the flags change |

**TABLE 5–18** Bit test instructions

| *Instruction* | *Comment* |
|---|---|
| BT | Test a bit in the destination operand specified by the source operand |
| BTC | Test and complement a bit in the destination operand specified by the source operand |
| BTR | Test and reset a bit in the destination operand specified by the source operand |
| BTS | Test and set a bit in the destination operand specified by the source operand |

The remaining three bit test instructions also place the bit under test into the carry flag and change the bit under test. The BTC AX,4 instruction complements bit position 4 after testing it; the BTR AX,4 instruction clears it (0) after the test; and the BTS AX,4 instruction sets it (1) after the test.

## NOT and NEG

Logical inversion or the *one's complement* (NOT) and arithmetic sign inversion or the *two's complement* (NEG) are the last two logic functions presented in this section of the text. These are two of a few instructions that contain only one operand. Table 5–19 lists some variations of the NOT and NEG instructions. As with most other instructions, NOT and NEG can use any addressing mode except segment-register addressing.

The NOT instruction inverts all bits of a byte, word, or doubleword. The NEG instruction two's complements a number, which means that the arithmetic sign of a signed number changes from positive to negative or negative to positive. The NOT function is considered a logical operation and the NEG function is considered an arithmetic operation.

**TABLE 5–19** NOT and NEG instructions

| *Instruction* | *Comment* |
|---|---|
| NOT CH | CH is one's complemented |
| NEG CH | CH is two's complemented |
| NEG AX | AX is two's complemented |
| NOT EBX | EBX is one's complemented |
| NEG ECX | ECX is two's complemented |
| NOT TEMP | The contents of the data-segment memory location addressed by TEMP are one's complemented (the size of TEMP is determined by how TEMP is defined) |
| NOT BYTE PTR [BX] | The byte contents of the data-segment memory location addressed by BX are one's complemented |

## 5–5 SHIFTS AND ROTATES

Shift and rotate instructions manipulate binary numbers at the binary bit level, as did the AND, OR, Exclusive-OR, and NOT instructions. Shifts and rotates find their most common application in low-level software used to control I/O devices. The 8086–80486 and Pentium microprocessors contain a complete set of shift and rotate instructions used to shift or rotate any memory data or register.

### Shifts

Shift instructions position or move numbers to the left or right within a register or memory location. They also perform simple arithmetic such as multiplication by powers of $2^{+n}$ (*left shift*) and division by powers of $2^{-n}$ (*right shift*). The microprocessor's instruction set contains four different shift instructions: two are *logical shifts* and two are *arithmetic shifts*. All four shift operations appear in Figure 5–9.

In Figure 5–9 there are two right shifts and two left shifts. The logical shifts move a 0 into the right-most bit position for a logical left shift, and a 0 into the left-most bit position for a logical right shift. There are also two arithmetic shifts. The arithmetic and logical left shifts are identical. The arithmetic and logical right shifts are different because the arithmetic right shift copies the sign bit through the number, while the logical right shift copies a 0 through the number.

Logical shift operations function with unsigned numbers and arithmetic shifts function with signed numbers. Logical shifts multiply or divide unsigned data, and arithmetic shifts multiply or divide signed data. A shift left always multiplies by 2 for each bit position shifted, and a shift right always divides by 2 for each bit position shifted. Shifting a number two places multiplies or divides by 4.




**FIGURE 5–9** The shift instructions showing the operation and direction of the shift.

Table 5–20 illustrates some addressing modes allowed for the various shift instructions. There are two different forms of shifts that allow any register (except the segment register) or memory location to be shifted. One mode uses an immediate shift count, and the other uses register CL to hold the shift count. Note that CL must hold the shift count. When CL is the shift count, it does not change when the shift instruction executes. Note that the shift count is a modulo-32 count. This means that a shift count of 33 will shift the data one place (33 / 32 = remainder of 1).

**EXAMPLE 5–28**

```
0000 C1 E2 0E                   SHL  DX,14

                                  or

0003 B1 0E                      MOV  CL,14
0005 D3 E2                      SHL  DX,CL
```

Example 5–28 shows how to shift the DX register left 14 places in two different ways. The first method uses an immediate shift count of 14. The second method loads a 14 into CL and then uses CL as the shift count. Both instructions shift the contents of the DX register logically to the left 14 binary bit positions or places.

**EXAMPLE 5–29**

```
                        ;Multiply AX by 10 (1010)
                        ;
0000 D1 E0                      SHL  AX,1               ;AX times 2
0002 8B D8                      MOV  BX,AX
0004 C1 E0 02                   SHL  AX,2               ;AX times 8
0007 03 C3                      ADD  AX,BX              ;10 times AX
                        ;
                        ;Multiply AX by 18 (10010)
                        ;
0009 D1 E0                      SHL  AX,1               ;AX times 2
000B 8B D8                      MOV  BX,AX
000D C1 E0 03                   SHL  AX,3               ;AX times 16
0010 03 D8                      ADD  BX,AX              ;18 times AX
```

**TABLE 5–20** Shift instructions

| *Instruction* | *Comment* |
|---|---|
| SHL AX,1 | Logically shift AX left one place |
| SHR BX,12 | Logically shift BX right 12 places |
| SHR ECX,10 | Logically shift ECX right 10 places |
| SAL DATA1,CL | Arithmetically shift DATA1, in the data segment, left the number of places contained in CL |
| SAR SI,2 | Arithmetically shift SI right 2 places |
| SAR EDX,14 | Arithmetically shift EDX right 14 places |

*Note*: The 8086/8088 microprocessor allows an immediate shift count of 1 only.

Suppose that the contents of AX must be multiplied by 10, as in Example 5–29. This can be done in two ways: by the MUL instruction or by shifts and additions. A number is doubled when it shifts one binary place to the left. When a number is doubled, then added to the number times 8, the result is 10 times the number. The number 10 decimal is 1010 in binary. A logic 1 appears in both the 2s and 8s positions. If 2 times the number is added to 8 times the number, the result is 10 times the number. Using this technique, a program can be written to multiply by any constant. This technique often executes faster than the multiply instruction in earlier versions of the Intel microprocessor.

*Double-Precision Shifts (80386/80486/Pentium Only).* The 80386/80486/Pentium contains two double-precision shifts, SHLD (*shift left*) and SHRD (*shift right*). Each instruction contains three operands instead of the two operands found with the other shift instructions. Both instructions function with two 16- or 32-bit registers or with one 16- or 32-bit memory location and a register.

The SHRD AX,BX,12 instruction is an example of the double-precision shift right instruction. This instruction logically shifts AX right by 12 bit positions. The right-most 12 bits of BX shift into the left-most 12 bits of AX. The contents of BX remain unchanged by this instruction. The shift count can be an immediate count, as in this example, or can be found in register CL like other shift instructions.

The SHLD EBX,ECX,16 instruction shifts EBX left. The left-most 16 bits of ECX fill the right-most 16 bits of EBX after the shift. As before, the contents of ECX, the second operand, remain unchanged. This instruction, as well as SHRD, affects the flag bits.

## Rotates

Rotate instructions position binary data by rotating the information in a register or memory location either from one end to another or through the carry flag. They are often used to shift or position numbers that are wider than 16 bits in the 8086–80286 microprocessor or wider than 32 bits in the 80386/80486/Pentium. The four rotate instructions available appear in Figure 5–10.

Numbers rotate through a register or a memory location and the C flag (carry). They also rotate through a register or memory location without carry. With either type of rotate instruction, the programmer can select either a left or a right rotate. Addressing modes used with rotate are the same as used with the shifts. A rotate count can be immediate or located in register CL. Table 5–21 lists some of the possible rotate instructions. If CL is used for a rotate count, it does not change. The count in CL is a modulo-32 count.

**EXAMPLE 5–30**

```
0000 D1 E0                      SHL   AX,1
0002 D1 D3                      RCL   BX,1
0004 D1 D2                      RCL   DX,1
```

Rotates are often used to shift wide numbers to the left or right. The program listed in Example 5–30 shifts the 48-bit number in registers DX, BX, and AX left one binary place. Notice that the least-significant 16 bits (AX) are shifted left first. This moves the

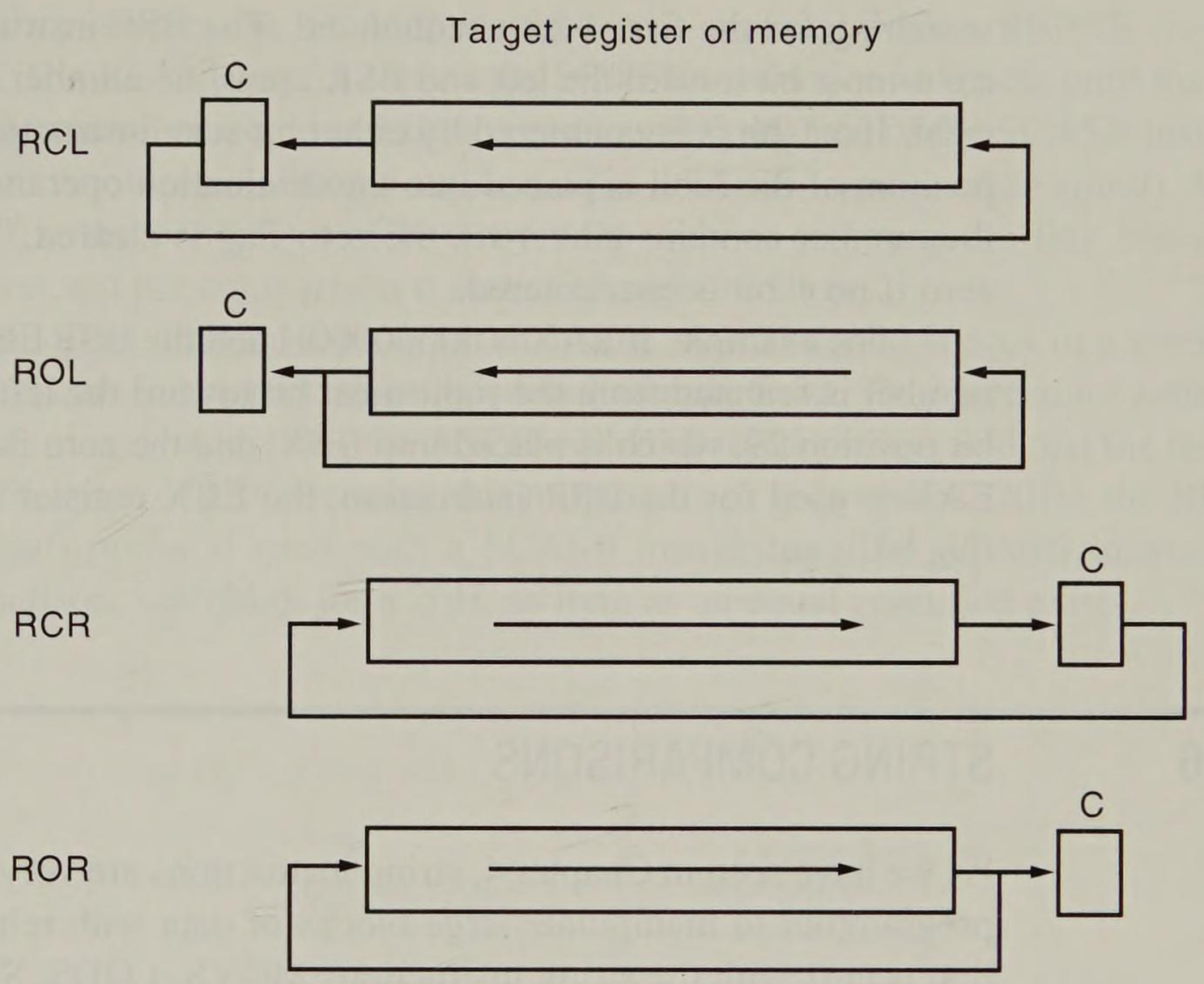



**FIGURE 5-10** The rotate instructions showing the direction and operation of each rotate.

**TABLE 5–21** Rotate instructions

| *Instruction* | *Comment* |
|---|---|
| ROL SI,4 | SI rotates left four places |
| RCL BL,6 | BL rotated left, through carry, six places |
| ROL ECX,18 | ECX rotates left 18 places |
| RCR AH,CL | AH rotates right, through carry, the number of places contained in CL |
| ROR WORD PTR [BP],2 | The word contents of the stack-segment memory location addressed by BP rotates right two places |

*Note*: The 8086/8088 microprocessor can only use an immediate rotate count of 1.

left-most bit of AX into the carry flag bit. Next, the rotate BX instruction rotates carry into BX and its left-most bit moves into carry. The last instruction rotates carry into DX and the shift is complete.

## Bit Scan Instructions

Although the bit scan instructions don't shift or rotate numbers, they do scan through a number searching for a 1-bit. Because this is accomplished by shifting the number, bit scan instructions are included in this section of the text.

The bit scan instructions BSF (*bit scan forward*) and BSR (*bit scan reverse*) are only available in the 80386/80486/Pentium microprocessor. Both forms scan through a number

searching for the first 1-bit encountered. The BSF instruction scans the number from the right-most bit toward the left and BSR scans the number from the left-most bit toward the right. If a 1-bit is encountered by either bit scan instruction, the zero flag is set and the bit position of the 1-bit is placed into the destination operand. If no 1-bit is encountered (e.g. the number contains all zeros), the zero flag is cleared. This means that the result is not-zero if no 1-bit is encountered.

For example, if EAX = 60000000H and the BSF EBX,EAX instruction executes, the number is scanned from the right-most bit toward the left. The first 1-bit encountered is at bit position 29, which is placed into EBX, and the zero flag bit is set. If the same value for EAX is used for the BSR instruction, the EBX register is loaded with a 30 and the zero flag bit is set.

## 5–6 STRING COMPARISONS

As we have seen in Chapter 4, string instructions are very powerful because they allow the programmer to manipulate large blocks of data with relative ease. Block data manipulation occurs with the string instructions MOVS, LODS, STOS, INS, and OUTS.

In this section, we discuss additional string instructions that allow a section of memory to be tested against a constant or against another section of memory. To accomplish these tasks we use the SCAS (*string scan*) or CMPS (*string compare*) instructions.

### SCAS

The string scan instruction (SCAS) compares the AL register with a byte block of memory, the AX register with a word block of memory, or the EAX register with a doubleword block of memory (80386/80486/Pentium only). The SCAS instruction subtracts memory from AL, AX, or EAX without affecting either the register or the memory location. The opcode used for byte comparison is *SCASB*, the opcode used for the word comparison is *SCASW*, and the opcode used for a doubleword comparison is *SCASD*. In all cases the contents of the extra-segment memory location addressed by DI is compared with AL, AX, or EAX. Recall that this default segment (ES) cannot be changed with a segment override prefix.

Like the other string instructions, SCAS instructions use the direction flag (D) to select either the auto-increment or auto-decrement operation for DI. They also repeat if prefixed by a *conditional* repeat prefix.

**EXAMPLE 5–31**

```
0000 BF 0011 R          MOV   DI,OFFSET BLOCK ;address data
0003 FC                 CLD                   ;auto-increment
0004 B9 0064            MOV   CX,100          ;load counter
0007 32 C0              XOR   AL,AL           ;clear AL
0009 F2/AE              REPNE SCASB           ;search
```

Suppose a section of memory is 100 bytes in length and begins at location BLOCK. This section of memory must be tested to see if any location contains a 00H. The program in Example 5–31 shows how to search this part of memory for a 00H using the SCASB

instruction. In this example, the SCASB instruction is prefixed with a REPNE (*repeat while not equal*). The REPNE prefix causes the SCASB instruction to repeat until the CX register reaches 0, or until an equal condition exists as the outcome of the SCASB instruction's comparison. Another conditional repeat prefix is REPE (*repeat while equal*). With either repeat prefix, the contents of CX decrement without affecting the flag bits. The SCASB instruction and the comparison it makes change the flags.

Suppose you must develop a program that skips ASCII-coded spaces in a memory array. This task appears in the procedure listed in Example 5–32. This procedure assumes that the DI register already addresses the ASCII-coded character string, and that the length of the string is 256 bytes or less. Because this program is to skip spaces (20H), the REPE (repeat while equal) prefix is used with a SCASB instruction. The SCASB instruction repeats the comparison, searching for a 20H, as long as an equal condition exists.

**EXAMPLE 5–32**

```
0000                    SKIP    PROC FAR

0000 FC                         CLD                     ;auto-increment
0001 B9 0100                    MOV  CX,256             ;counter
0004 B0 20                      MOV  AL,20H             ;get space
0006 F3/AE                      REPE SCASB              ;search
0008 CB                         RET

0009                    SKIP    ENDP
```

## CMPS

The *compare strings* instruction (CMPS) always compares two sections of memory data as bytes (CMPSB), words (CMPSW), or doublewords (CMPSD). Note that only the 80386/80486/Pentium can use doublewords. The contents of the data-segment memory location addressed by SI are compared with the contents of the extra-segment memory location addressed by DI. The CMPS instruction increments or decrements both SI and DI. The CMPS instruction is normally used with either the REPE or REPNE prefix. Alternatives to these prefixes are REPZ (*repeat while zero*) and REPNZ (*repeat while not zero*), but usually we use REPE or REPNE.

**EXAMPLE 5–33**

```
0000                    MATCH   PROC FAR

0000 BE 0075 R                  MOV  SI,OFFSET LINE  ;address LINE
0003 BF 007F R                  MOV  DI,OFFSET TABLE ;address TABLE
0006 FC                         CLD                  ;auto-increment
0007 B9 000A                    MOV  CX,10           ;counter
000A F3/A6                      REPE CMPSB           ;search
000C CB                         RET

000D                    MATCH   ENDP
```

Example 5–33 illustrates a short procedure that compares two sections of memory searching for a match. The CMPSB instruction is prefixed with a REPE. This causes the search to continue as long as an equal condition exists. When the CX register

becomes 0, or an unequal condition exists, the CMPSB instruction stops execution. After the CMPSB instruction ends, if the CX register is zero or the flags indicate an equal condition, the two strings match. If CX is not zero or the flags indicate a not-equal condition, the strings do not match.

## 5–7 SUMMARY

1. Addition (ADD) can be 8-, 16-, or 32-bit. The ADD instruction allows any addressing mode except segment-register addressing. Most flags (C, A, S, Z, P, and O) change when the ADD instruction executes. A different type of addition, add-with-carry (ADC), adds two operands and the contents of the carry flag bit (C). The 80486/Pentium microprocessor has an additional instruction (XADD) that combines an addition with an exchange.
2. The increment instruction (INC) adds one to a byte, word, or doubleword contents of a register or memory location. The INC instruction affects the same flag bits as ADD except the carry flag. The BYTE PTR, WORD PTR, and DWORD PTR directives appear with the INC instruction when the contents of a memory location are addressed by a pointer.
3. Subtraction (SUB) is a byte, word, or doubleword and is performed on a register or a memory location. The only form of addressing not allowed by the SUB instruction is segment-register addressing. The subtract instruction affects the same flags as ADD and subtracts the carry if the SBB form is used.
4. The decrement (DEC) instruction subtracts one from the contents of a register or a memory location. The only addressing modes not allowed with DEC are immediate or segment-register addressing. The DEC instruction does not affect the carry flag and is often used with BYTE PTR, WORD PTR, or DWORD PTR.
5. The compare (CMP) instruction is a special form of subtraction that does not store the difference; instead the flags change to reflect the difference. Compare is used to compare an entire byte or word located in any register (except segment) or memory location. An additional compare instruction, CMPXCHG, is a combination of compare and exchange instructions in the 80486/Pentium microprocessor.
6. Multiplication is byte, word, or doubleword and can be signed (IMUL) or unsigned (MUL). The 8-bit multiplication always multiplies register AL by an operand with the product found in AX. The 16-bit multiplication always multiplies register AX by an operand with the product found in DX–AX. The 32-bit multiplication always multiplies register EAX by an operand with the product found in EDX–EAX. A special IMUL immediate instruction that contains three operands exists on the 80286–80486 and Pentium. For example, the IMUL BX,CX,3 instruction multiplies CX by 3 and leaves the product in BX.
7. Division is byte, word, or doubleword and can be signed (IDIV) or unsigned (DIV). For 8-bit division, the AX register divides by the operand, after which the quotient appears in AL and the remainder in AH. In 16-bit division, the DX–AX register divides by the operand, after which the quotient appears in AX and the remainder in

DX. In 32-bit division, the EDX–EAX register is divided by the operand, after which the quotient appears in EAX and the remainder in EDX. Note that the remainder after a signed division always assumes the sign of the dividend.

8. BCD data add or subtract in packed form by adjusting the result of the addition with DAA or the subtraction with DAS. ASCII data are added, subtracted, multiplied, or divided when the operations are adjusted with AAA, AAS, AAM, and AAD.
9. The AAM instruction can also convert a binary number into unpacked BCD. This instruction converts a binary number between 00H–63H into unpacked BCD in AX. The AAM instruction divides AX by 10, leaving the quotient in AH and the remainder in AL.
10. The AND, OR, and Exclusive-OR instructions perform logic functions on a byte, word, or doubleword stored in a register or memory location. All flags change with these instructions, with carry (C) and overflow (O) cleared.
11. The TEST instruction performs the AND operation, but the logical product is lost. This instruction changes the flag bits to indicate the outcome of the test.
12. The NOT and NEG instructions perform logical inversion and arithmetic inversion. The NOT instruction one's complements an operand and the NEG instruction two's complements an operand.
13. There are eight different shift and rotate instructions. Each of these instructions shifts or rotates a byte, word, or doubleword register or memory data. These instructions have two operands: the first is the location of the data shifted or rotated, and the second is an immediate shift rotate count, or CL. If the second operand is CL, the CL register holds the shift or rotate count. In the 80386/80486/Pentium microprocessor two additional double-precision shifts (SHRD and SHLD) instructions exist.
14. The scan string (SCAS) instruction compares AL, AX, or EAX with the contents of the extra-segment memory location addressed by DI.
15. The string compare (CMPS) instruction compares the byte, word, or doubleword contents of two sections of memory. One section is addressed by DI, in the extra segment, and the other by SI, in the data segment.
16. The SCAS and CMPS instructions repeat with the REPE or REPNE prefixes. The REPE prefix repeats the string instruction while an equal condition exists. The REPNE prefix repeats the string instruction while a not-equal condition exists.

---

## 5–8 QUESTIONS AND PROBLEMS

1. Select an ADD instruction that will:
   (a) add BX to AX
   (b) add 12H to AL
   (c) add EDI and EBP
   (d) add 22H to CX
   (e) add the data addressed by SI to AL
   (f) add CX to the data stored at memory location FROG
2. What is wrong with the ADD ECX,AX instruction?

3. Is it possible to add CX to DS with the ADD instruction?
4. If AX = 1001H and DX = 20FFH, list the sum and the contents of each flag register bit (C, A, S, Z, and O) after the ADD AX,DX instruction executes.
5. Develop a short sequence of instructions that adds AL, BL, CL, DL, and AH. Save the sum in the DH register.
6. Develop a short sequence of instructions that adds AX, BX, CX, DX, and SP. Save the sum in the DI register.
7. Develop a short sequence of instructions that adds ECX, EDX, and ESI. Save the sum in the EDI register.
8. Select an instruction that adds BX to DX and also adds the contents of the carry flag (C) to the result.
9. Choose an instruction that adds a 1 to the contents of the SP register.
10. What is wrong with the INC [BX] instruction?
11. Select a SUB instruction that will:
    (a) subtract BX from CX
    (b) subtract 0EEH from DH
    (c) subtract DI from SI
    (d) subtract 3322H from EBP
    (e) subtract the data addressed by SI from CH
    (f) subtract the data stored 10 words after the location addressed by SI from DX
    (g) subtract AL from memory location FROG
12. If DL = 0F3H and BH = 72H, list the difference after BH subtracts from DL and show the contents of the flag register bits.
13. Write a short sequence of instructions that subtracts the numbers in DI, SI, and BP from the AX register. Store the difference in register BX.
14. Choose an instruction that subtracts one from register EBX.
15. Explain what the SBB [DI – 4],DX instruction accomplishes.
16. Explain the difference between the SUB and CMP instructions.
17. When two 8-bit numbers are multiplied, where is the product found?
18. When two 16-bit numbers are multiplied, what two registers hold the product? Show which register contains the most-significant and least-significant portions of the product.
19. When two numbers multiply, what happens to the O and C flag bits?
20. Where is the product stored for the MUL EDI instruction?
21. What is the difference between the IMUL and MUL instructions?
22. Write a sequence of instructions that cubes the 8-bit number found in DL. Load DL with a 5 initially and make sure your result is a 16-bit number.
23. Describe the operation of the IMUL BX,DX,100H instruction.
24. When 8-bit numbers are divided, in which register is the dividend found?
25. When 16-bit numbers are divided, in which register is the quotient found?
26. What type of errors may be detected during a division?
27. Explain the difference between the IDIV and DIV instructions.
28. Where is the remainder found after an 8-bit division?
29. Write a short sequence of instructions that divides the number in BL by the number in CL and then multiply the result by two. The final answer must be a 16-bit number stored in the DX register.

30. Which instructions are used with BCD arithmetic operations?
31. Which instructions are used with ASCII arithmetic operations?
32. Explain how the AAM instruction converts from binary to BCD.
33. Develop a sequence of instructions that converts the unsigned number in AX (values of 0–65535) into a 5-digit BCD number stored in memory beginning at the location addressed by the BX register in the data segment. Note that the most-significant character is stored first and no attempt is made to blank leading zeros.
34. Develop a sequence of instructions that adds the 8-digit BCD number in AX and BX to the 8-digit BCD number in CX and DX. (AX and CX are the most-significant registers. The result must be found in CX and DX after the addition.)
35. Select an AND instruction that will:
    (a) AND BX with DX and save the result in BX
    (b) AND 0EAH with DH
    (c) AND DI with BP and save the result in DI
    (d) AND 1122H with EAX
    (e) AND the data addressed by BP with CX and save the result in memory
    (f) AND the data stored in four words before the location addressed by SI with DX and save the result in DX
    (g) AND AL with memory location WHAT and save the result at location WHAT
36. Develop a short sequence of instructions that clears (0) the three left-most bits of DH without changing the remainder DH and store the result in BH.
37. Select an OR instruction that will:
    (a) OR BL with AH and save the result in AH
    (b) OR 88H with ECX
    (c) OR DX with SI and save the result in SI
    (d) OR 1122H with BP
    (e) OR the data addressed by BX with CX and save the result in memory
    (f) OR the data stored 40 bytes after the location addressed by BP with AL and save the result in AL
    (g) OR AH with memory location WHEN and save the result in WHEN
38. Develop a short sequence of instructions that sets (1) the right-most five bits of DI without changing the remaining bits of DI. Save the result in SI.
39. Select the XOR instruction that will:
    (a) XOR BH with AH and save the result in AH
    (b) XOR 99H with CL
    (c) XOR DX with DI and save the result in DX
    (d) XOR 1A23H with ESP
    (e) XOR the data addressed by EBX with DX and save the result in memory
    (f) XOR the data stored 30 words after the location addressed by BP with DI and save the result in DI
    (g) XOR DI with memory location WELL and save the result in DI
40. Develop a sequence of instructions that sets (1) the right-most four bits of AX, clears (0) the left-most three bits of AX, and inverts bits 7, 8, and 9 of AX.
41. Describe the difference between the AND and TEST instructions.
42. Select an instruction that tests bit position 2 of register CH.
43. What is the difference between the NOT and a NEG instructions?

44. Select the correct instruction to perform each of the following tasks:
    (a) shift DI right 3 places with zeros moved into the left-most bit
    (b) move all bits in AL left one place, making sure that a 0 moves into the right-most bit position
    (c) rotate all the bits of AL left 3 places
    (d) rotate carry right one place through EDX
    (e) move the DH register right one place making sure that the sign of the result is the same as the sign of the original number
45. What does the SCASB instruction accomplish?
46. For string instructions, DI always addresses data in the _____ segment.
47. What is the purpose of the D flag-bit?
48. Explain what the REPE prefix does when coupled with the SCASB instruction.
49. What condition or conditions will terminate the repeated string instruction REPNE SCASB?
50. Describe what the CMPSB instruction accomplishes.
51. Develop a sequence of instructions that scans through a 300H byte section of data-segment memory called LIST, searching for a 66H.

# CHAPTER 6

# Program Control Instructions

## INTRODUCTION

The program control instructions direct the flow of a program and allow the flow to change. A change in flow often occurs after a decision, made with the CMP or TEST instruction, is followed by a conditional jump instruction. This chapter explains the program control instructions including the jumps, calls, returns, interrupts, and machine control instructions.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use both conditional and unconditional jump instructions to control the flow of a program.
2. Use the call and return instructions to include procedures in the program structure.
3. Explain the operation of the interrupts and interrupt control instructions.
4. Use machine control instructions to modify the flag bits.
5. Use ENTER and LEAVE to enter and leave programming structures.

## 6–1 THE JUMP GROUP

The main program control instruction, jump (JMP), allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction. A conditional jump instruction allows the programmer to make decisions based upon numerical tests. The results of these numerical tests are held in the flag bits, which are then tested by conditional jump instructions. Another instruction similar to the conditional jump, the conditional set, is explained with the conditional jump instructions in this section.

In this section of the text, we cover all jump instructions and illustrate their use with sample programs. We also revisit the LOOP and conditional LOOP instructions, first presented in Chapter 3, because they are also forms of the jump instruction.

## Unconditional Jump (JMP)

Three types of unconditional jump instructions are available to the microprocessor: short jump, near jump, and far jump (refer to Figure 6–1). The *short jump* is a 2-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes from the address following the jump. The 3-byte *near jump* allows a branch or jump within ±32K bytes (or anywhere in the current code segment) from the instruction in the current code segment. Remember, the segments are cyclic in nature, which means that one location above offset address FFFFH is offset address 0000H. For this reason, if you jump two bytes ahead in memory and the instruction pointer addresses offset address FFFFH, the flow continues at offset address 0001H. Thus a displacement of ±32K bytes allows a jump to any location within the current code segment. Finally, the 5-byte *far jump* allows a jump to any memory location within the entire real-memory system. The short and near jumps are often called *intrasegment jumps* and the far jumps are often called *intersegment jumps*.

In the 80386/80486/Pentium microprocessor, the near jump is within ±2G if the machine is operated in the protected mode with a code segment of 4G bytes in length, and ±32K bytes if operated in the real mode. The protected-mode 80386/80486/Pentium jump uses a 32-bit displacement that is not shown in Figure 6–1. The 80386/80486/Pentium far jump allows a jump to any location within the 4G byte address range of these microprocessors.

*Short Jump.* Short jumps are called *relative jumps* because they can be moved, along with their related software, to any location in current code segment without a change. This is because the jump address is *not* stored with the opcode. Instead of a jump address, a distance or *displacement* follows the opcode. The short jump displacement is a *distance* represented by a 1-byte signed number whose value ranges between +127 and –128. The short jump instruction appears in Figure 6–2. When the microprocessor executes a short jump,

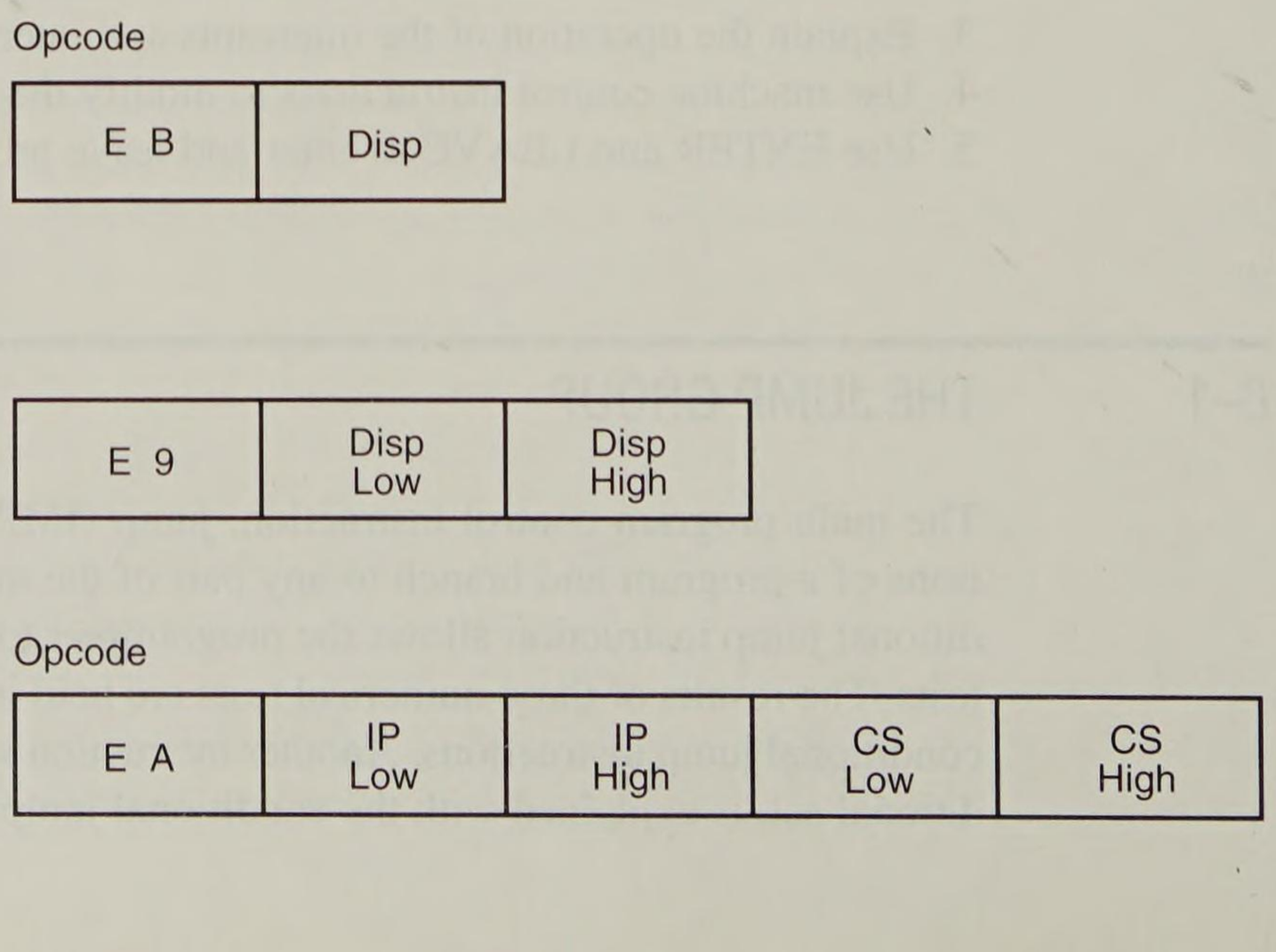



**FIGURE 6–1** The three forms of the JMP instruction. (Note that Disp is either an 8- or 16-bit signed displacement or distance.)

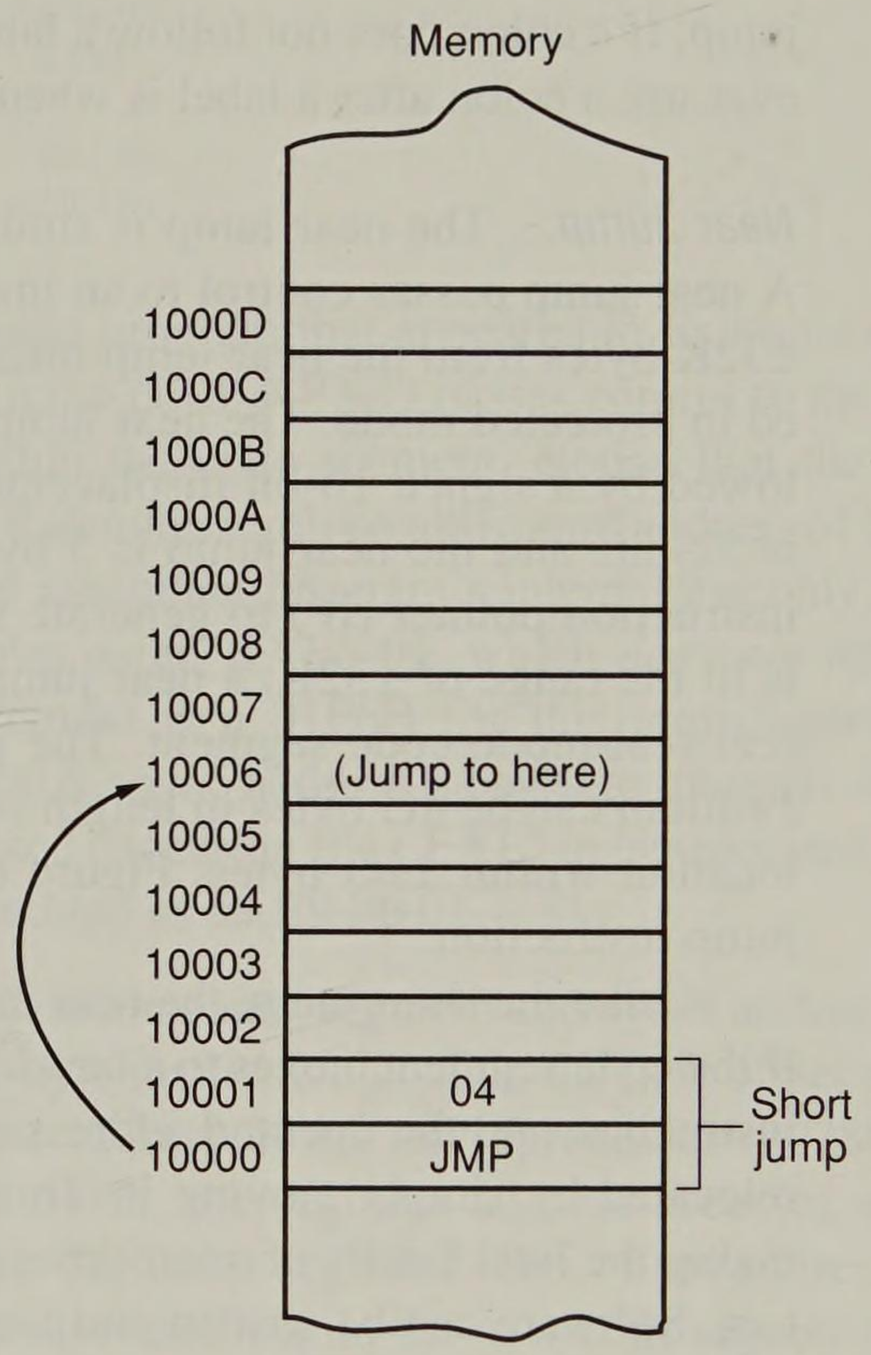



**FIGURE 6–2** A short JMP to four memory locations beyond the address of the next instruction. (Note that this will skip the next four bytes of memory.)

the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment. The short jump instruction branches to this new address for the next instruction in the program.

**EXAMPLE 6–1**

```
0000 33 DB                    XOR   BX,BX

0002 B8 0001          START:  MOV   AX,1
0005 03 C3                    ADD   AX,BX
0007 EB 17                    JMP   SHORT NEXT

0020 8B D8            NEXT:   MOV   BX,AX
0022 EB DE                    JMP   START
```

Example 6–1 shows how short jump instructions pass control from one part of the program to another. It also illustrates the use of a label with the jump instruction. Notice how one jump (JMP SHORT NEXT) uses the SHORT directive to force a short jump, while the other does not. *Most* assembler programs choose the best form of the jump instruction so the second jump instruction (JMP START) also assembles as a short jump. If we add the address of the next instruction (0009H) to the sign-extended displacement of the first jump (0017H), the address of NEXT is at location 0017H + 0009H or 0020H.

Whenever a jump instruction references an address, a *label* normally identifies the address. JMP NEXT is an example, which jumps to label NEXT for the next instruction. We hardly ever use an actual hexadecimal address with any jump instruction. The label NEXT must be followed by a colon (NEXT:) to allow an instruction to reference it for a

jump. If a colon does not follow a label, you *cannot* jump to it. Note that the only time we ever use a colon after a label is when the label is used with a jump or call instruction.

*Near Jump.* The near jump is similar to the short jump except the distance is farther. A near jump passes control to an instruction in the current code segment located within ±32K bytes from the near jump instruction or ±2G in the 80386/80486/Pentium operated in protected mode. The near jump is a 3-byte instruction that contains an opcode followed by a signed 16-bit displacement. In the 80386/80486/Pentium, the displacement is 32-bits and the near jump is 5 bytes in length. The signed displacement adds to the instruction pointer (IP) to generate the jump address. Because the signed displacement is in the range of ±32K, a near jump can jump to *any* memory location within the current real-mode code segment. The protected-mode code segment in the 80386/80486/Pentium can be 4G bytes in length so the 32-bit displacement allows a near jump to any location within ±2G bytes. Figure 6–3 illustrates the operation of the real-mode near jump instruction.

Like the short jump, the near jump is called a *relative jump* because it is relocatable. If the code segment moves to a new location in the memory, the distance between the jump instruction and the operand address remains the same. This allows a code segment to be relocated by simply moving it. This feature, along with the relocatable data segments, makes the Intel family of microprocessors ideal for use in a general-purpose computer system. Software can be written and loaded anywhere in the memory and it functions without modification because of the relative jumps and relocatable data segments. This is true of very few other microprocessors.

**EXAMPLE 6–2**

```
0000 33 DB                      XOR   BX,BX

0002 B8 0001            START:  MOV   AX,1
```

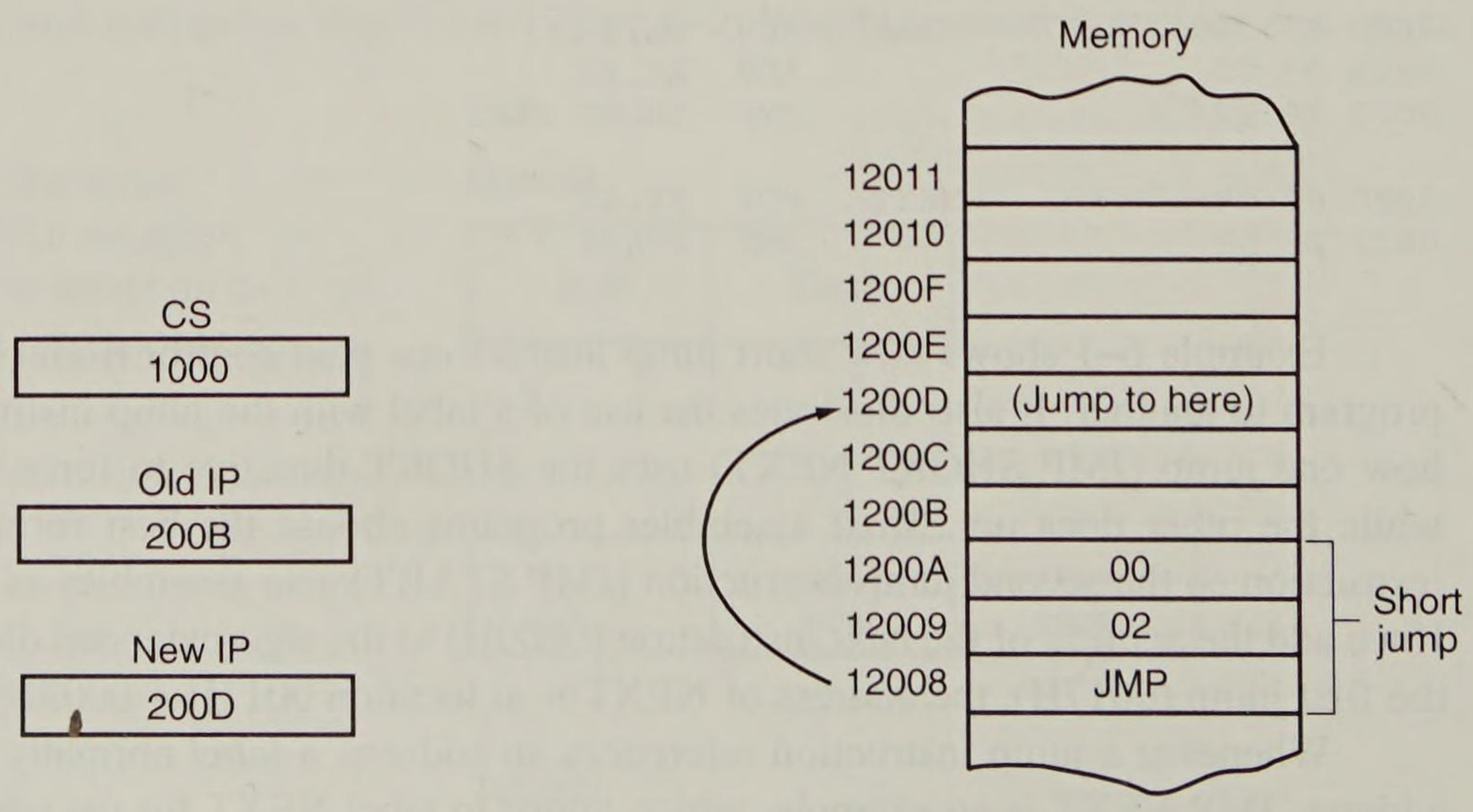



**FIGURE 6–3** A near JMP that adds the displacement (0002H) to the contents of IP.

```
0005 03 C3                      ADD   AX,BX
0007 E9 0200 R                  JMP   NEXT

0200 8B D8              NEXT:   MOV   BX,AX
0202 E9 0002 R                  JMP   START
```

Example 6–2 shows the same basic program that appeared in Example 6–1, except the jump distance is greater. The first jump (JMP NEXT) passes control to the instruction at offset-memory location 0200H within the code segment. Notice that the instruction assembles as an E9 0200 R. The letter R denotes a relocatable jump address of 0200H. The relocatable address of 0200H is for the assembler program's internal use only. The actual machine language instruction assembles as an E9 F6 01, which *does not* appear in the assembler listing. The actual displacement is a 01F6H for this jump instruction. The assembler lists the jump address as 0200 R so the address is easier to interpret as we develop software. If we were to view the linked execution file (.EXE) in hexadecimal, we would see that this jump instruction has assembled as an E9 F6 01.

*Far Jump.* Far jumps (see Figure 6–4) obtain a new segment and offset address to accomplish the jump. Bytes 2 and 3 of this 5-byte instruction contain the new offset address and bytes 4 and 5 contain the new segment address. If the microprocessor (80286–80486 or Pentium) is operated in the protected mode, the segment address accesses a descriptor which contains the base address of the far jump segment. The offset address, which is either 16- or 32-bits, contains the offset location within the new code segment.

**EXAMPLE 6–3**

```
                        EXTRN  UP:FAR

0000 33 DB              XOR    BX,BX
```

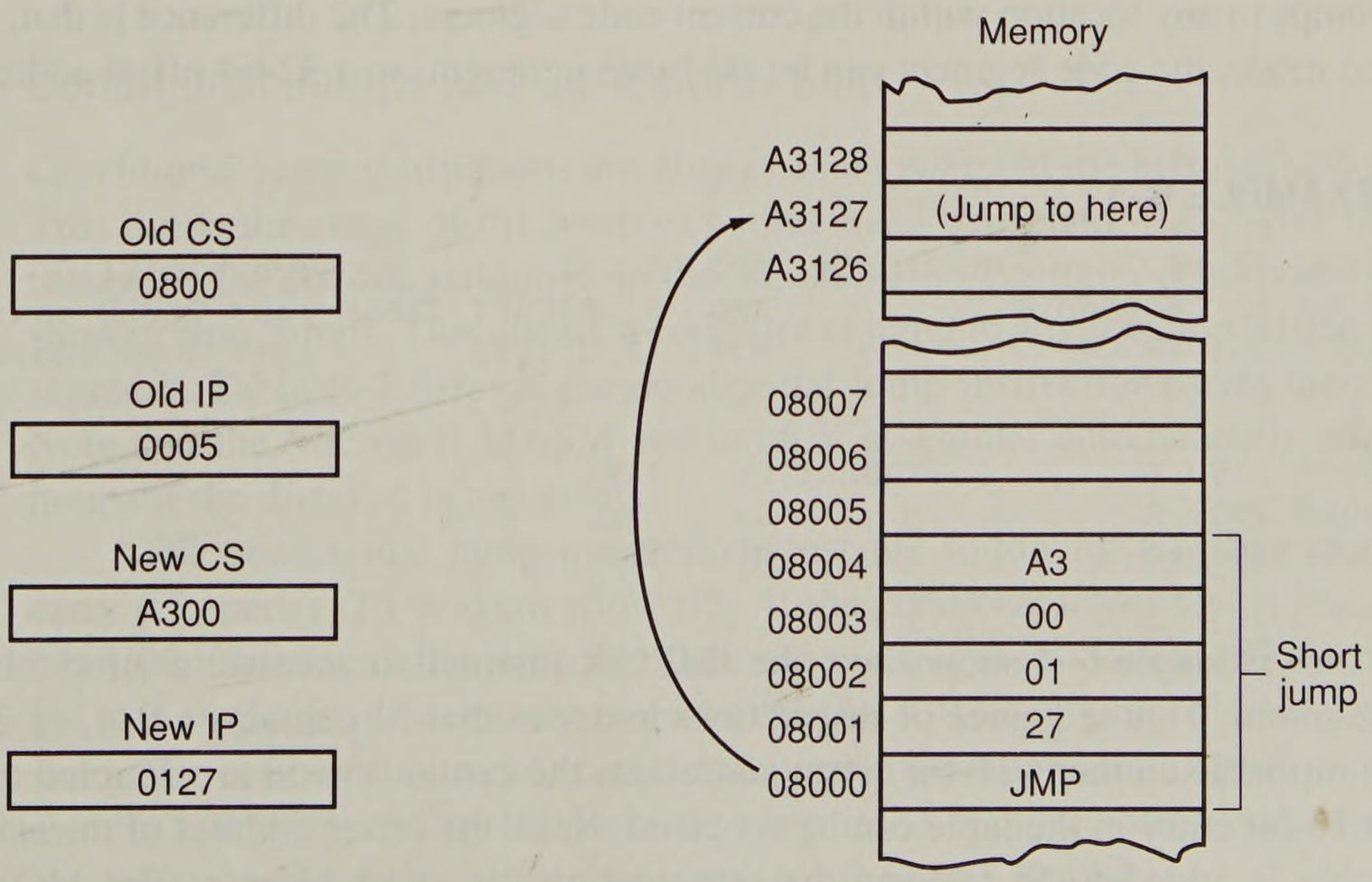



**FIGURE 6–4** A far JMP that replaces the contents of both the CS and IP registers with the four bytes that follow the opcode.

```
0002 B8 0001          START: MOV  AX,1
0005 03 C3                   ADD  AX,BX
0007 E9 0200 R               JMP  NEXT

0200 8B D8            NEXT:  MOV  BX,AX
0202 EA 0002 ---- R          JMP  FAR PTR START

0207 EA 0000 ---- E          JMP  UP
```

Example 6–3 lists a short program that uses a far jump instruction. The far jump instruction sometimes appears with the FAR PTR directive as illustrated. Another way to obtain a far jump is to define a label as a far label. A label is far only if it is external to the current code segment. The JMP UP instruction in the example references a far label. The label UP is defined as a far label by the EXTRN UP:FAR directive. External labels appear in programs that contain more than one program file.

When the program files are joined, the linker inserts the address for the UP label into the JMP UP instruction. It also inserts the segment address in the JMP START instruction. The segment address in JMP FAR PTR START is listed as ---- *R* for relocatable and the segment address in JMP UP is listed as ---- *E* for external. In both case the ---- is filled in by the linker when it links or joins the program files.

*Jumps with Register Operands.* The jump instruction can also specify a 16- or 32-bit register as an operand. This automatically sets up the instruction as an indirect jump. The address of the jump is in the register specified by the jump instruction. Unlike the displacement associated with the near jump, the contents of the register are transferred directly into the instruction pointer. The register contents do not add to the instruction pointer like the short and near jumps do. The JMP AX instruction, for example, copies the contents of the AX register into the IP register when the jump occurs. This allows a jump to any location within the current code segment. In the 80386/80486/Pentium, a JMP EAX instruction also jumps to any location within the current code segment. The difference is that, in the protected mode, the code segment can be 4G bytes in length, so a 32-bit offset address is needed.

**EXAMPLE 6–4**

```
0000 03 F6                ADD  SI,SI              ;double SI
0002 81 C6 000B R         ADD  SI,OFFSET TABLE ;add TABLE address
0006 2E: 8B 04            MOV  AX,CS:[SI]         ;get jump address
0009 FF E0                JMP  AX

000B 1000 R          TABLE: DW   ZERO
000D 2000 R                 DW   ONE
000F 3000 R                 DW   TWO
```

Example 6–4 shows how the JMP AX instruction accesses a jump table in the code segment. This sequence of instructions assumes that SI contains a 0, 1, or 2. Because the jump table contains 16-bit offset addresses, the contents of SI are doubled to 0, 2, or 4, so a 16-bit entry in the table can be accessed. Next, the offset address of the start of the jump table is added to SI to form the reference to the jump address. The MOV AX,CS:[SI] instruction then fetches an address from the code segment jump table so the JMP AX instruction jumps to the location stored in the jump table.

This example fetches address ZERO if a 0 appears in SI. Once the address is fetched, the JMP AX instruction causes the program to branch to address ZERO. The same is true for an initial value of 1 and 2 in the SI register. These numbers access addresses ONE and TWO for their branch locations.

*Indirect Jumps Using an Index.* The jump instruction may also use the [ ] form of addressing to directly access the jump table. The jump table can contain offset addresses for near indirect jumps or segment and offset addresses for far indirect jumps. (This type of jump is also known as a double-indirect jump if the register jump is called an indirect jump.) The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction. Example 6–5 uses the JMP CS:[SI] instruction instead of JMP AX. This reduces the length of the program.

**EXAMPLE 6–5**

```
0000 03 F6                    ADD  SI,SI              ;double SI
0002 81 C6 0009 R             ADD  SI,OFFSET TABLE ;add TABLE address
0006 2E: FF 24                JMP  CS:[SI]

0009 1000 R             TABLE: DW   ZERO
000B 2000 R                    DW   ONE
000D 3000 R                    DW   TWO
```

The mechanism used to access the jump table is identical with a normal memory reference. The JMP CS:[SI] instruction points to a jump address stored at the code segment offset location addressed by SI. It jumps to the address stored in the memory at this location. Both the register and indirect indexed jump instructions usually address a 16-bit offset. This means that both types of jumps are near jumps. If a JMP FAR PTR CS:[SI] instruction appears in a program, the microprocessor assumes that the jump table contains doubleword, 32-bit addresses (IP and CS).

## Conditional Jumps and Conditional Sets

Conditional jump instructions are always *short jumps* in the 8086–80286 microprocessor. This limits the range of the jump to within +127 bytes and –128 bytes from the location following the conditional jump. In the 80386/80486/Pentium, conditional jumps are either short or near jumps. This allows a conditional jump to any location within the current code segment. Table 6–1 lists all the conditional jump instructions with their test conditions. Note that the Microsoft MASM version 6.X assembler automatically adjusts conditional jumps if the distance is too great.

The conditional jump instructions test the following flag bits: sign (S), zero (Z), carry (C), parity (P), and overflow (O). If the condition under test is true, a branch to the label associated with the jump instruction occurs. If the condition is false, the next sequential step in the program executes. For example, a JC will jump if the carry bit is set.

The operation of most conditional jump instructions is straightforward because they usually test just one flag bit. Relative magnitude comparisons require more complicated conditional jump instructions that test more than one flag bit.

Because we use both signed and unsigned numbers, and the order of these numbers is different, there are two sets of conditional jump instructions for magnitude comparisons.

**TABLE 6–1** Conditional jump instructions

| *Instruction* | *Condition Tested* | *Comment* |
|---|---|---|
| JA | C = 0 and Z = 0 | Jump if above |
| JAE | C = 0 | Jump if above or equal |
| JB | C = 1 | Jump if below |
| JBE | C = 1 or Z = 1 | Jump if below or equal |
| JC | C = 1 | Jump if carry set |
| JE or JZ | Z = 1 | Jump if equal to or jump zero |
| JG | Z = 0 and S = 0 | Jump if greater than |
| JGE | S = 0 | Jump if greater than or equal |
| JL | S < > 0 | Jump if less than |
| JLE | Z = 1 or S <> 0 | Jump if less than or equal |
| JNC | C = 0 | Jump if carry cleared |
| JNE or JNZ | Z = 0 | Jump if not equal or jump not zero |
| JNO | O = 0 | Jump if no overflow |
| JNS | S = 0 | Jump if no sign |
| JNP/JPO | P = 0 | Jump if no parity/jump parity odd |
| JO | O = 1 | Jump if overflow is set |
| JP/JPE | P = 1 | Jump if parity/jump parity even |
| JS | S = 1 | Jump if sign is set |
| JCXZ | CX = 0 | Jump if CX = 0 |
| *JECXZ | ECX = 0 | Jump if ECX = 0 |

*Note*: * = 80386/80486/Pentium only.

Figure 6–5 shows the order of both signed and unsigned 8-bit numbers. The 16- and 32-bit numbers follow the same order as the 8-bit numbers except they are larger. Notice that an FFH (255) is above the 00H in the set of unsigned numbers, but an FFH (–1) is less than 00H for signed numbers. Therefore, an unsigned FFH is *above* 00H, but a signed FFH is *less than* 00H.

When we compare signed numbers, we use JG, JL, JGE, JLE, JE, and JNE. The terms *greater than* and *less than* refer to signed numbers. When we compare unsigned numbers, we use JA, JB, JAE, JBE, JE, and JNE. The terms *above* and *below* refer to unsigned numbers.

The remaining conditional jumps test individual flag bits such as overflow and parity. Notice that JE has an alternative opcode, JZ. All instructions have alternates, but many aren't used in programming because they don't make much sense. (The alternates appear in Appendix B with the instruction set listing.) For example, the JA instruction (jump if above) has the alternative JNBE (jump if not below or equal). A JA functions exactly like a JNBE, but JNBE sounds and is awkward when compared to JA.

The most radical of the conditional jump instructions are JCXZ (jump if CX = 0) and JECXZ (jump if ECX = 0). These are the only conditional jump instructions that do not test the flag bits. Instead, JCXZ directly tests the contents of the CX register without affecting the flag bits and JECXZ tests the contents of the ECX register. For the JCXZ instruction, if CX = 0, a jump occurs, and if CX <> 0, no jump occurs. For the JECXZ instruction, if ECX = 0, a jump occurs, and if CX <> 0, no jump occurs.

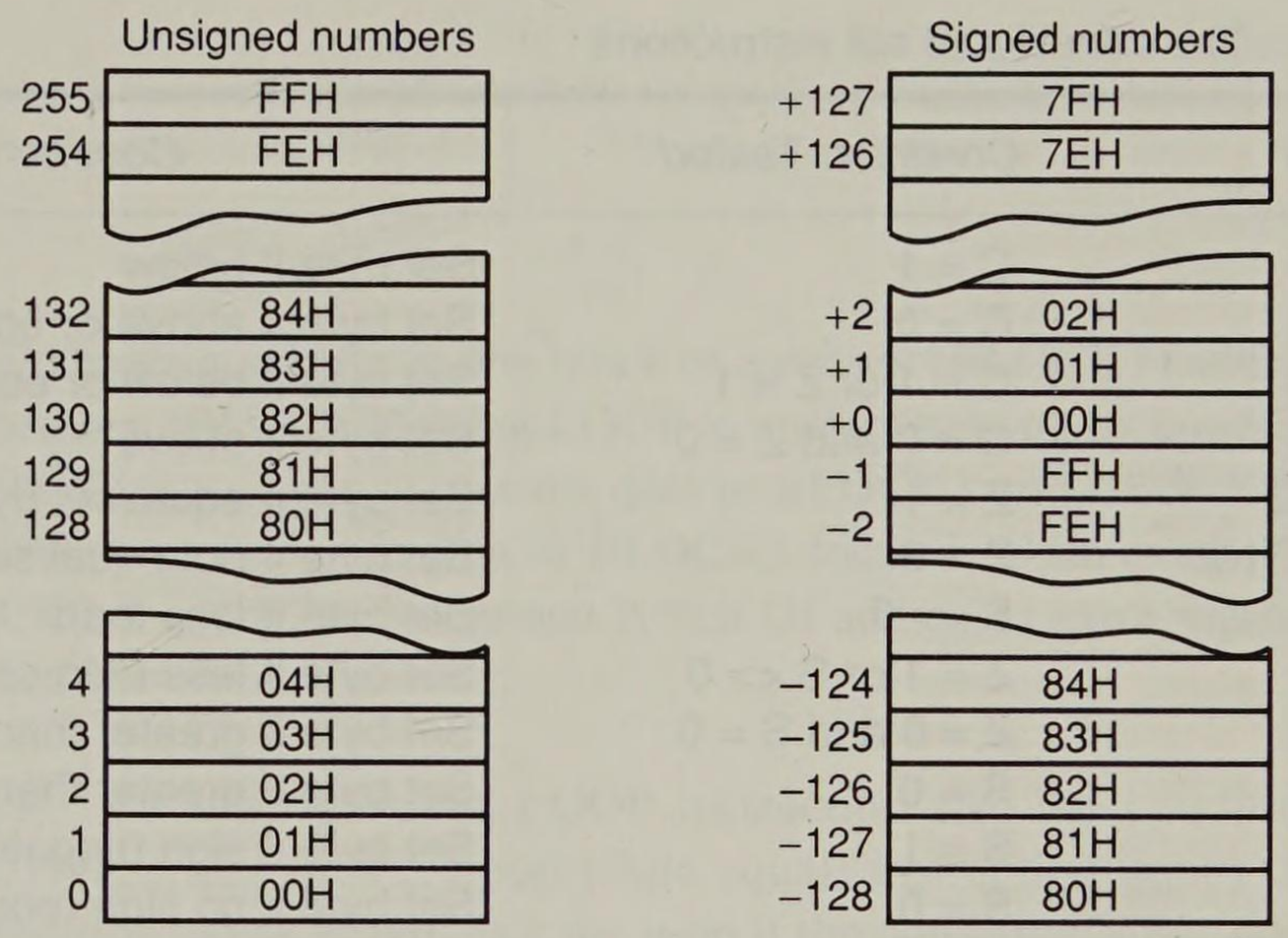



**FIGURE 6–5** Signed and unsigned numbers follow different orders.

**EXAMPLE 6–6**

```
                        ;A procedure that searches a table of 100 bytes for 0AH
                        ;
0000                    SCAN    PROC  NEAR

0000 BE 0000 R                  MOV   SI,OFFSET TABLE ;address TABLE
0003 B9 0064                    MOV   CX,100          ;load count
0006 B0 0A                      MOV   AL,0AH          ;load search data
0008 FC                         CLD
0009 F2/AE                      REPNE SCASB
000B E3 23                      JCXZ  NOT_FOUND
000D C3                         RET

000E                    SCAN    ENDP
```

A program that uses JCXZ appears in Example 6–6. Here the SCASB instruction searches a table for a 0AH. Following the search, a JCXZ instruction tests CX to see if the count has reached zero. If the count is zero, the 0AH is not found in the table. Another method used to test to see if the data are found is the JNE instruction. If JNE replaces JCXZ, it performs the same function. After the SCASB instruction executes, the flags indicate a not-equal condition if the data was not found in the table.

*The Conditional Set Instructions.* In addition to the conditional jump instructions, the 80386/80486/Pentium microprocessor also contains conditional set instructions. The conditions tested by conditional jumps are put to work with the conditional set instructions. The conditional set instructions set a byte to either a 01H or clear a byte to 00H depending on the outcome of the condition under test. Table 6–2 lists the available forms of the 80386/80486/Pentium conditional set instructions.

These instructions are useful where a condition must be tested at a point much later in the program. For example, a byte can be set to indicate that the carry has cleared some point in the program by using the SETNC MEM instruction. This instruction places a 01H into memory location MEM if carry is cleared and a 00H into MEM if carry is set. The

**TABLE 6–2** The conditional set instructions

| *Instruction* | *Condition Tested* | *Comment* |
|---|---|---|
| SETB | C = 1 | Set byte if below |
| SETAE | C = 0 | Set byte if above or equal |
| SETBE | C = 1 or Z = 1 | Set byte if below or equal |
| SETA | C = 0 and Z = 0 | Set byte if above |
| SETE/SETZ | Z = 1 | Set byte if equal/set byte if zero |
| SETNE/SETNZ | Z = 0 | Set byte if not equal/set byte if not zero |
| SETL | S <> 0 | Set byte if less than |
| SETLE | Z = 1 or S <> 0 | Set byte if less than or equal |
| SETG | Z = 0 and S = 0 | Set byte if greater than |
| SETGE | S = 0 | Set byte if greater than or equal |
| SETS | S = 1 | Set byte if sign (negative) |
| SETNS | S = 0 | Set byte if no sign (positive) |
| SETC | C = 1 | Set byte if carry |
| SETNC | C = 0 | Set byte if no carry |
| SETO | O = 1 | Set byte if overflow |
| SETNO | O = 0 | Set byte if no overflow |
| SETP | P = 1 | Set byte if parity (even) |
| SETNP | P = 0 | Set byte if no parity (odd) |

contents of MEM can be tested at a later point in the program to determine if carry is cleared at the point where the SETNC MEM instruction executed.

## LOOP

The LOOP instruction is a combination of a decrement CX and the JNZ conditional jump. In the 8086–80286, LOOP decrements CX, and if CX < > 0, it jumps to the address indicated by the label. If CX becomes a 0, the next sequential instruction executes. In the 80386/80486/Pentium, LOOP decrements either CX or ECX depending upon the instruction mode. If the 80386/80486/Pentium is operated in the 16-bit instruction mode, LOOP uses CX; if operated in the 32-bit instruction mode LOOP uses ECX. This default can be changed by the LOOPW (using CX) and LOOPD (using ECX) instructions in the 80386/80486/Pentium only.

**EXAMPLE 6–7**

```
                        ;A procedure that adds word in BLOCK1 to BLOCK2
                        ;
0000                    ADDS    PROC NEAR

0000 B9 0064                    MOV  CX,100             ;load count
0003 BE 0064 R                  MOV  SI,OFFSET BLOCK1   ;address BLOCK1
0006 BF 0000 R                  MOV  DI,OFFSET BLOCK2   ;address BLOCK2
0009                    AGAIN:
0009 AD                         LODSW                   ;get BLOCK1 data
000A 26: 03 05                  ADD  AX,ES:[DI]         ;add BLOCK2 data
```

```
000D AB                         STOSW                   ;store in BLOCK2
000E E2 F9                      LOOP AGAIN              ;repeat 100 times
0010 C3                         RET

0011                    ADDS    ENDP
```

Example 6–7 shows how data in one block of memory (BLOCK1) adds to data in a second block of memory (BLOCK2) using LOOP to control how many numbers add. The LODSW and STOSW instructions access the data in BLOCK1 and BLOCK2. The ADD AX,ES:[DI] instruction accesses the data in BLOCK2 located in the extra segment. The only reason BLOCK2 is in the extra segment is that DI addresses extra-segment data for the STOSW instruction.

*Conditional LOOPs.* As with REP, the LOOP instruction also has conditional forms: LOOPE and LOOPNE. The LOOPE (loop while equal) instruction jumps if CX < > 0 while an equal condition exists. It will exit the loop if the condition is not-equal or if the CX register decrements to 0. The LOOPNE (loop while not-equal) instruction jumps if CX < > 0 while a not-equal condition exists. It will exit the loop if the condition is equal or if the CX register decrements to 0. In the 80386/80486/Pentium, the conditional LOOP instruction can use either CX or ECX as the counter. LOOPEW or LOOPED or LOOPNEW or LOOPNED override the instruction mode if needed.

As with the conditional repeat instructions, alternates exist for LOOPE and LOOPNE. The LOOPE instruction is the same as LOOPZ and LOOPNE is the same as LOOPNZ. In most programs only the LOOPE and LOOPNE apply.

## 6–2 PROCEDURES

The procedure or *subroutine* is an important part of any computer system's architecture. A *procedure* is a group of instructions that usually performs one task. A *procedure* is a reusable section of the software that is stored in memory once, but used as often as necessary. This saves memory space and makes it easier to develop software. The only disadvantage of a procedure is that it takes the computer a small amount of time to link to the procedure and return from it. The CALL instruction links to the procedure and the RET (*return*) instruction returns from the procedure.

The stack stores the return address whenever a procedure is called during the execution of a program. The CALL instruction pushes the address of the instruction following the CALL (*return address*) on the stack. The RET instruction removes an address from the stack so the program returns to the instruction following the CALL.

**EXAMPLE 6–8**

```
0000                    SUMS    PROC NEAR

0000 03 C3                      ADD  AX,BX
0002 03 C1                      ADD  AX,CX
0004 03 C2                      ADD  AX,DX
```

```
0006 C3                         RET

0007                    SUMS    ENDP

0007                    SUMS1   PROC FAR

0007 03 C3                      ADD  AX,BX
0009 03 C1                      ADD  AX,CX
000B 03 C2                      ADD  AX,DX
000D CB                         RET

000E                    SUMS1   ENDP

000E                    SUMS2   PROC NEAR  USES BX CX DX

0011 03 C3                      ADD  AX,BX
0013 03 C1                      ADD  AX,CX
0015 03 C2                      MOV  AX,DX
                                RET

001B                    SUMS2   ENDP
```

With the assembler, there are specific rules for the storage of procedures. A procedure begins with the PROC directive and ends with the ENDP directive. Each directive appears with the name of the procedure. This programming structure makes it easy to locate the procedure in a program listing. The PROC directive is followed by the type of procedure: NEAR or FAR. Example 6–8 shows how the assembler uses the definition of both a near (intrasegment) and far (intersegment) procedure. In MASM version 6.X, the NEAR or FAR type can be followed by the USES statement. The USES statement allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure. The USES statement is also illustrated in Example 6–8.

When we compare these two procedures, the only difference is the opcode of the return instruction. The near return instruction uses opcode C3H and the far return instruction uses opcode CBH. A near return removes a 16-bit number from the stack and places it into the instruction pointer to return from the procedure in the current code segment. A far return removes a 32-bit number from the stack and places it into both IP and CS to return from the procedure to any memory location.

Procedures that are to be used by all software (*global*) should be written as far procedures. Procedures that are used by a given task (*local*) are normally defined as near procedures.

## CALL

The CALL instruction transfers the flow of the program to the procedure. The CALL instruction differs from the jump instruction because a CALL saves a return address on the stack. The return address returns control to the instruction that immediately follows the CALL in a program when a RET instruction executes.

*Near CALL.* The near CALL instruction is three bytes long, with the first byte containing the opcode and the second and third bytes containing the displacement or distance of ±32K in the 8086–80286. This is identical to the form of the near jump instruction. The

80386/80486/Pentium uses a 32-bit displacement when operated in the protected mode to allow a distance of ±2G bytes. When the near CALL executes, it first pushes the offset address of the next instruction on the stack. The offset address of the next instruction appears in the instruction pointer (IP or EIP). After saving this return address, it then adds the displacement from bytes 2 and 3 to the IP to transfer control to the procedure. There is no short CALL instruction.

Why save the IP or EIP on the stack? The instruction pointer always points to the next instruction in the program. For the CALL instruction, the contents of IP/EIP are pushed onto the stack so program control passes to the instruction following the CALL after a procedure ends. Figure 6–6 shows the return address (IP) stored on the stack, and the call to the procedure for the 8086–80286.

*Far CALL.* The far CALL instruction is like a far jump because it can CALL a procedure stored in any memory location in the system. The far CALL is a 5-byte instruction that contains an opcode followed by the next value for the IP and CS registers. Bytes 2 and 3 contain the new contents of the IP and bytes 4 and 5 contain the new contents for CS.

The far CALL instruction places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2–5 of the instruction. This allows the far CALL to call a procedure located anywhere in the memory and return from that procedure.

Figure 6–7 shows how the far CALL instruction calls a far procedure. Here the contents of IP and CS are pushed onto the stack. Next the program branches to the procedure.

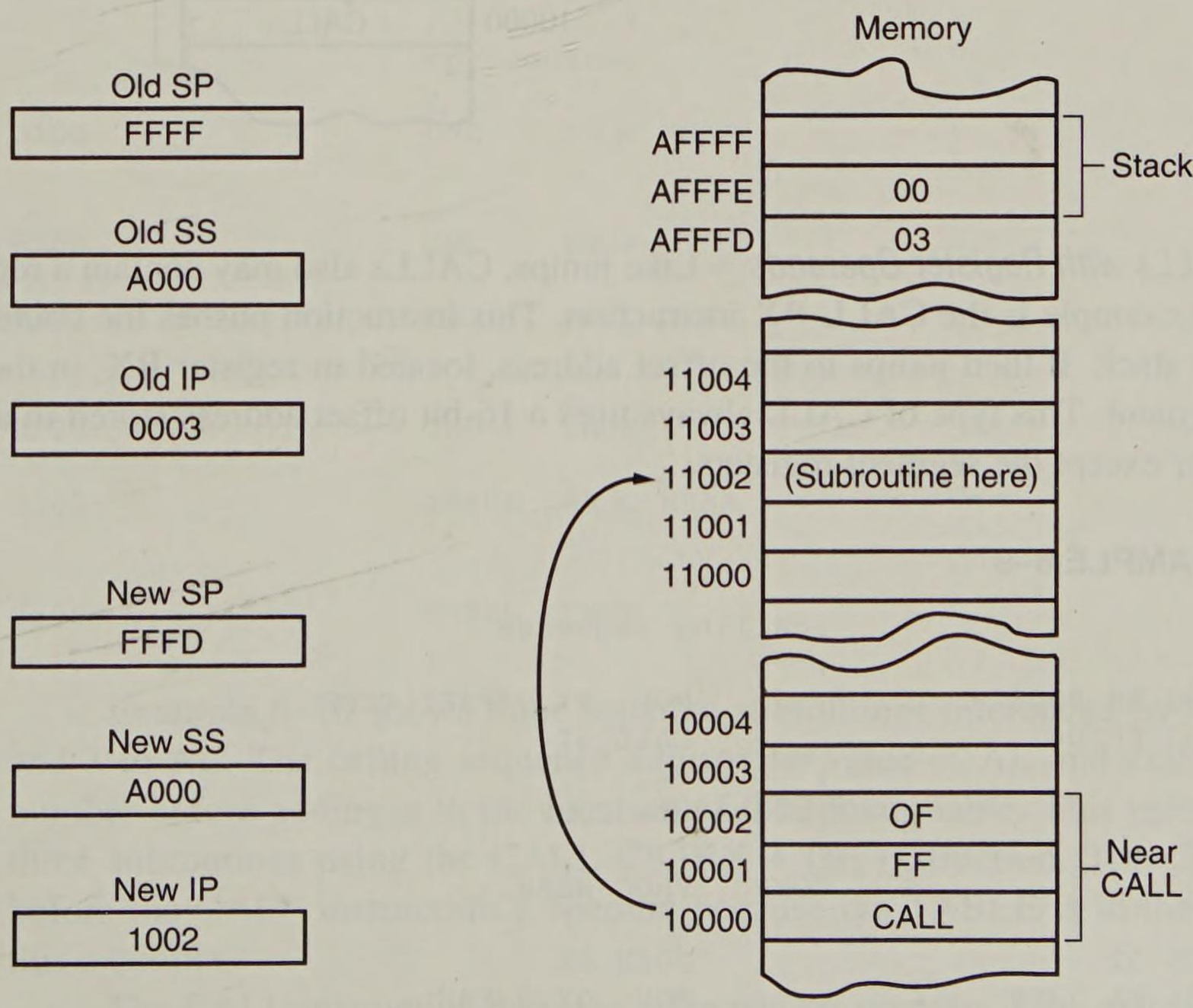



**FIGURE 6–6** The effect of a near call instruction on the stack and the SP, SS, and IP registers. Notice how the old IP is stored on the stack.

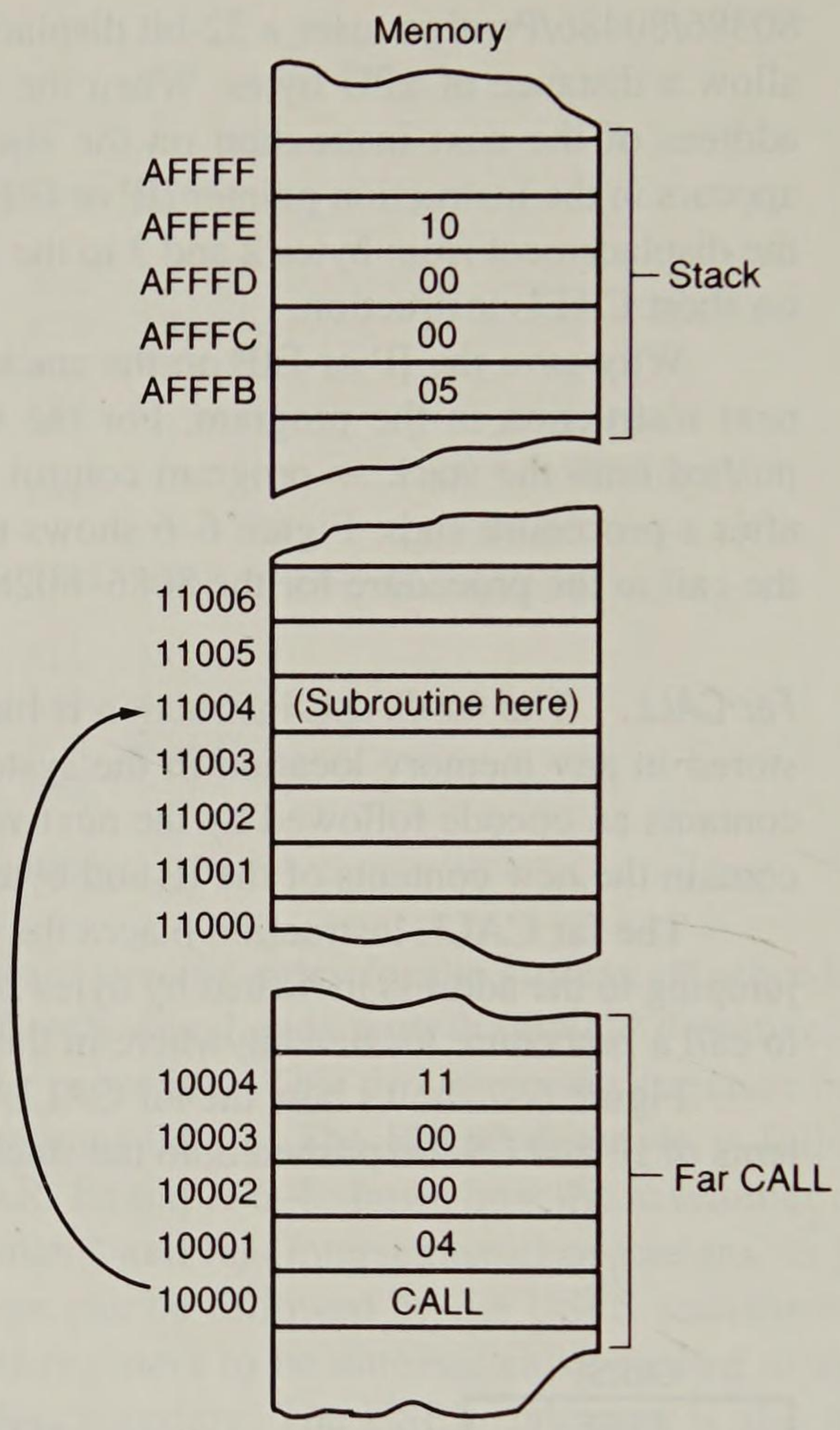



**FIGURE 6–7** The far CALL instruction.

*CALLs with Register Operands.* Like jumps, CALLs also may contain a register operand. An example is the CALL BX instruction. This instruction pushes the contents of IP onto the stack. It then jumps to the offset address, located in register BX, in the current code segment. This type of CALL always uses a 16-bit offset address stored in any 16-bit register except the segment registers.

**EXAMPLE 6–9**

```
                        ;calling sequence
                        ;
0000 BE 0005 R                  MOV  SI,OFFSET COMP
0003 FF D6                      CALL SI
                        ;
                        ;procedure COMP
                        ;
0005                    COMP    PROC NEAR

0005 52                         PUSH DX
0006 BA 03F8                    MOV  DX,03F8H
0009 EC                         IN   AL,DX
000A 42                         INC  DX
000B EE                         OUT  DX,AL
```

```
000C 5A                         POP   DX
000D C3                         RET

000E                    COMP    ENDP
```

Example 6–9 illustrates the use of the CALL register instruction to call a procedure that begins at offset address COMP. The OFFSET address COMP moves into the SI register and then the CALL SI instruction calls the procedure beginning at address COMP.

*CALLs with Indirect Memory Addresses.* The CALL with an indirect memory address is particularly useful whenever we need to choose different subroutines in a program. This selection process is often keyed with a number that addresses a CALL address in a lookup table.

**EXAMPLE 6–10**

```
                        ;lookup table
                        ;
0000 0100 R             TABLE   DW    ONE
0002 0200 R                     DW    TWO
0004 0300 R                     DW    THREE
                        ;
                        ;calling sequence
                        ;
0006 4B                         DEC   BX                ;scale BX
0007 03 DB                      ADD   BX,BX             ;double BX
0009 BF 0000 R                  MOV   DI,OFFSET TABLE   ;address TABLE
000C 2E: FF 11                  CALL  CS:[BX+DI]
                        ;
                        ;procedures
                        ;
0100                    ONE     PROC NEAR
                          .       .
                          .       .
0100                    ONE     ENDP

0200                    TWO     PROC NEAR
                          .       .
                          .       .
0200                    TWO     ENDP

0300                    THREE   PROC NEAR
                          .       .
                          .       .
0300                    THREE   ENDP
```

Example 6–10 shows three separate subroutines referenced by the numbers 1, 2, and 3 in AL. The calling sequence adjusts the value of AL and extends it to a 16-bit number before adding it to the location of the lookup table. This references one of the three subroutines using the CALL CS:[BX + DI] instruction. The CS: prefix appears before the CALL instruction's operand because the TABLE is in the code segment in this example.

The CALL instruction can also reference far pointers if the instruction appears as a CALL FAR PTR [SI]. This instruction retrieves a 32-bit address from the data-segment memory location addressed by SI and uses it as the address of a far procedure.

## RET

The return instruction (RET) removes either a 16-bit number (near return) from the stack and places it into IP or a 32-bit number (far return) and places it into IP and CS. The near and far return instructions are both defined in the procedure's PROC directive. This automatically selects the proper return instruction. In the 80386/80486/Pentium operated in the protected mode, the far return removes 6 bytes from the stack—the first four contain the new value for EIP, and the last two contain the new value for CS. The 80386/80486/Pentium protected-mode near return removes 4 bytes from the stack and places them into EIP.

When IP/EIP or IP/EIP and CS are changed, the address of the next instruction is at a new memory location. This new location is the address of the instruction that immediately follows the most recent CALL to a procedure. Figure 6–8 shows how the CALL instruction links to a procedure and how the RET instruction returns in the 8086–80286 microprocessor.

There is one other form of the return instruction. This form adds a number to the contents of the stack pointer (SP) before the return. If the pushes must be deleted before a return, a 16-bit displacement adds to the SP before the return retrieves the return address from the stack. The effect of this is to delete stack data, or skip stack data.

**EXAMPLE 6–11**

```
0000                    TESTS  PROC NEAR
0000 50                        PUSH AX
```

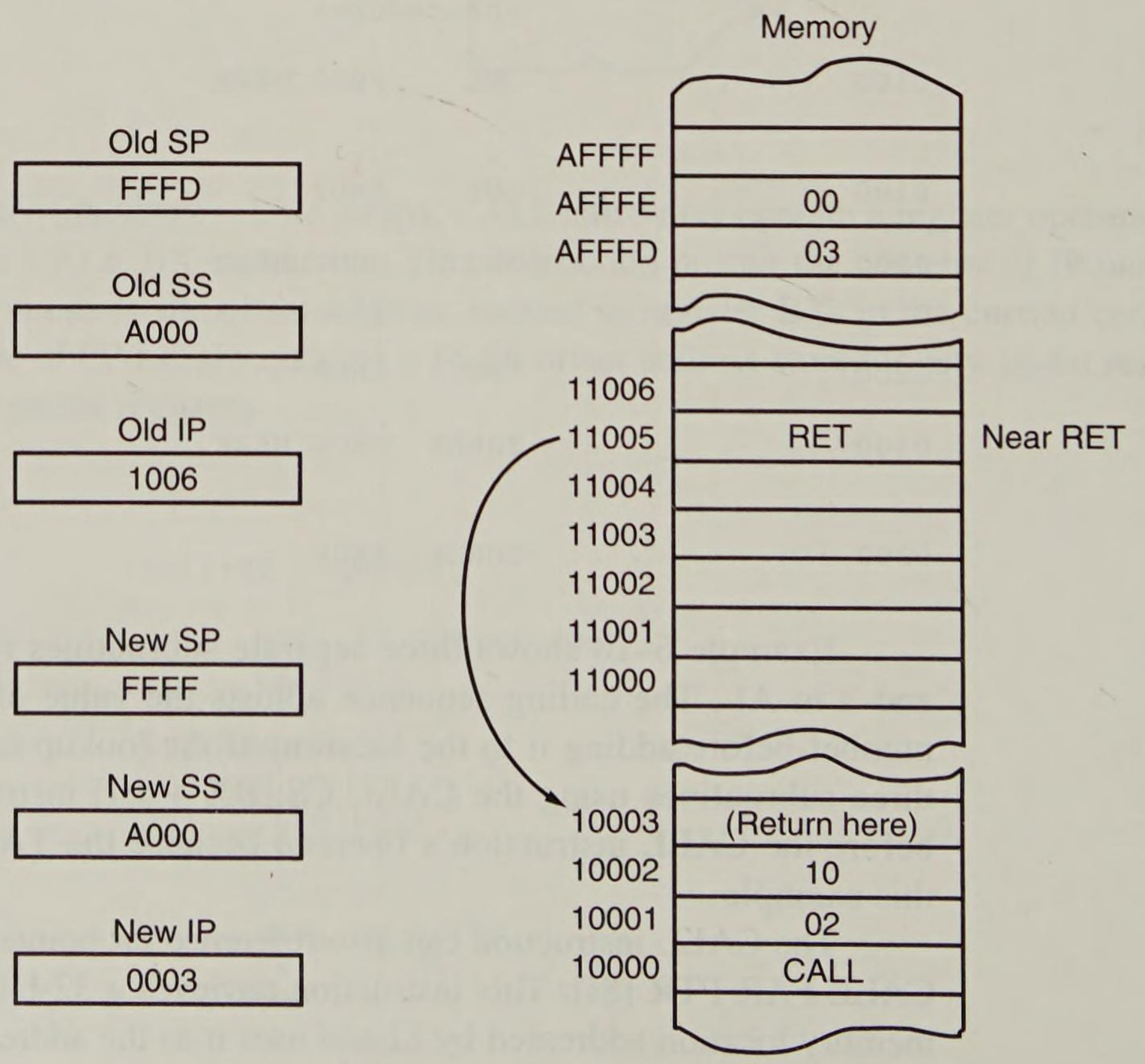



**FIGURE 6–8** The effect of a near RET instruction on the stack and the SP, SS, and IP registers.

```
0001 53                         PUSH BX
                                  .    .
                                  .    .
0030 C2 0004                    RET  4

0033                    TESTS   ENDP
```

Example 6–11 shows how this type of return erases the data placed on the stack by a few pushes. The RET 4 adds a 4 to SP before removing the return address from the stack. Since the PUSH AX and PUSH BX together place four bytes of data on the stack, this return effectively deletes AX and BX from the stack. This type of return appears rarely in assembly language programs, but is used in higher level programs to clear stack data before a return.

## 6–3 INTRODUCTION TO INTERRUPTS

An *interrupt* is either a hardware-generated CALL (externally derived from a hardware signal) or a software-generated CALL (internally derived from the execution of an instruction or by some other internal event). Either type interrupts the program by calling an *interrupt service procedure* or *interrupt handler*.

This section explains software interrupts, which are special types of CALL instructions in the 8086–80486 and Pentium microprocessors. We cover the three types of software interrupt instructions (INT, INTO, and INT 3), provide a map of the interrupt vectors, and explain the purpose of the special interrupt return instruction (IRET).

### Interrupt Vectors

An *interrupt vector* is a 4-byte number stored in the first 1,024 bytes of the memory (000000H–0003FFH) when the microprocessor operates in the real mode. In the protected mode, the vector table is replaced by a interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts. There are 256 different interrupt vectors. Each vector contains the address of an interrupt service procedure. Table 6–3 lists the interrupt vectors with a brief description and the memory location of each vector for the real mode. Each vector contains a value for IP and CS that forms the address of the interrupt service procedure. The first 2 bytes contain the IP and the last 2 bytes contain the CS.

Intel reserves the first 32 interrupt vectors for the 8086–80486/Pentium and future products. The remaining interrupt vectors (32–255) are available for the user. Some of the reserved vectors are for errors that occur during the execution of software, such as the divide error interrupt. Some vectors are reserved for the coprocessor. Still others occur for normal events in the system. In a personal computer, the reserved vectors are used for system functions as detailed later in this section. Vectors 1–6, 7, 9, 16, and 17 function in the real mode and protected mode; the remaining vectors function only in the protected mode.

**TABLE 6–3** Interrupt vectors

| *Number* | *Address* | *Microprocessor* | *Function* |
|---|---|---|---|
| 0 | 0H–3H | 8086–80486/Pentium | Divide error |
| 1 | 4H–7H | 8086–80486/Pentium | Single step |
| 2 | 8H–BH | 8086–80486/Pentium | NMI (hardware interrupt) |
| 3 | CH–FH | 8086–80486/Pentium | Breakpoint |
| 4 | 10H–13H | 8086–80486/Pentium | Interrupt on overflow |
| 5 | 14H–17H | 80286–80486/Pentium | BOUND interrupt |
| 6 | 18H–1BH | 80286–80486/Pentium | Invalid opcode |
| 7 | 1CH–1FH | 80286–80486/Pentium | Coprocessor emulation interrupt |
| 8 | 20H–23H | 80386–80486/Pentium | Double fault |
| 9 | 24H–27H | 80386 | Coprocessor segment overrun |
| 10 | 28H–2BH | 80386–80486/Pentium | Invalid task state segment |
| 11 | 2CH–2FH | 80386–80486/Pentium | Segment not present |
| 12 | 30H–33H | 80386–80486/Pentium | Stack fault |
| 13 | 34H–37H | 80386–80486/Pentium | General protection fault |
| 14 | 38H–3BH | 80386–80486/Pentium | Page fault |
| 15 | 3CH–3FH | | Reserved* |
| 16 | 40H–43H | 80286–80486/Pentium | Floating-point error |
| 17 | 44H–47H | 80486SX | Alignment check interrupt |
| 18 | 48H–4BH | Pentium | Machine check exception |
| 19–31 | 4CH–7FH | 8086–80486/Pentium | Reserved* |
| 32–255 | 80H–3FFH | 8086–80486/Pentium | User interrupts |

**Note*: Some of these interrupts will appear on newer versions of the 8086–80486/Pentium when they become available.

## Interrupt Instructions

The 8086–80486 has three different interrupt instructions available to the programmer: INT, INTO, and INT 3. In the real mode, each of these instructions fetches a vector from the vector table and then calls the procedure stored at the location addressed by the vector. In the protected mode, each of these instructions fetches an interrupt descriptor from the interrupt descriptor table. The descriptor specifies the address of the interrupt service procedure. The interrupt call is similar to a far CALL instruction because it places the return address (IP/EIP and CS) on the stack.

*INTs.* There are 256 different software interrupt instructions (INT) available to the programmer. Each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH). For example, the INT 100 uses interrupt vector 100, which appears at memory address 190H–193H. We calculate the address of the interrupt vector by multiplying the interrupt type number times four. For example, the INT 10H instruction calls the interrupt service procedure, whose address is stored beginning at memory location 40H (10H

× 4) in the real mode. In protected mode, the interrupt descriptor is located by multiplying the type number by 8 instead of 4, because each descriptor is eight bytes long.

Each INT instruction is two bytes in length. The first byte contains the opcode and the second byte contains the vector type number. The only exception to this is INT 3, a 1-byte special software interrupt used for breakpoints.

Whenever a software interrupt instruction executes, it (1) pushes the flags onto the stack, (2) clears the T and I flag bits, (3) pushes CS onto the stack, (4) fetches the new value for CS from the vector, (5) pushes IP/EIP onto the stack, (6) fetches the new value for IP/EIP from the vector, and (7) jumps to the new location addressed by CS and IP/EIP. The INT instruction performs just like a far CALL, except that it not only pushes CS and IP onto the stack, but it also pushes the flags onto the stack. The INT instruction is a combination PUSHF and far CALL instruction.

Notice that when the INT instruction executes, it clears the interrupt flag (I), which controls the external hardware interrupt input pin INTR (interrupt request). When I = 0, the microprocessor disables the INTR pin and when I = 1, the microprocessor enables the INTR pin.

Software interrupts are most commonly used to call system procedures. The system procedures are common to all system and application software. The interrupts often control printers, video displays, and disk drives. The INT instruction replaces a far CALL because the INT instruction is two bytes in length, while the far CALL is five bytes in length. Each time that the INT instruction replaces a far CALL, it saves three bytes of memory in a program. This can amount to a sizable savings if the INT instruction appears often in a program.

*IRET/IRETD.* The interrupt return instruction (IRET) is used only with software or hardware interrupt service procedures. Unlike a simple return instruction (RET), the IRET instruction will (1) pop stack data back into the IP, (2) pop stack data back into CS, and (3) pop stack data back into the flag register. The IRET instruction accomplishes the same tasks as the POPF and RET instructions.

Whenever an IRET instruction executes, it restores the contents of I and T from the stack. This is important because it preserves the state of these flag bits. If interrupts are enabled before an interrupt service procedure, they are *automatically* re-enabled by the IRET instruction because it restores the flag register.

In the 80286–80486 and Pentium, the IRETD instruction is used to return from an interrupt service procedure called in the protected mode. It differs from the IRET because it pops a 32-bit instruction pointer (EIP) from the stack. The IRET is used in the real mode and the IRETD is used in the protected mode.

*INT 3.* An INT 3 instruction is a special software interrupt designed to be used as a breakpoint. The difference between it and the other software interrupts is that INT 3 is a 1-byte instruction while the others are 2-byte instructions.

It is common to insert an INT 3 instruction in software to interrupt or break the flow of the software. This function is called a *breakpoint*. A breakpoint occurs for any software interrupt, but because INT 3 is one byte long, it is easier to use for this function. Breakpoints help to debug faulty software.

*INTO.* Interrupt on overflow (INTO) is a conditional software interrupt that tests the overflow flag (O). If O = 0, the INTO instruction performs no operation, but if O = 1 and an INTO instruction executes, an interrupt occurs via vector type number 4.

The INTO instruction appears in software that adds or subtracts signed binary numbers. With these operations it is possible to have an overflow. Either the JO instruction or INTO instruction detects the overflow condition.

*An Interrupt Service Procedure.* Suppose that, in a particular system, we must add the contents of DI, SI, BP, and BX and save the sum in AX. Because this is a common task in this system, it is worthwhile to develop the task as a software interrupt. Example 6–12 shows this software interrupt. The main difference between this procedure and a normal far procedure is that it ends with the IRET instruction instead of the RET instruction.

**EXAMPLE 6–12**

```
0000                    INTS    PROC FAR

0000 03 C3                      ADD  AX,BX
0002 03 C5                      ADD  AX,BP
0004 03 C7                      ADD  AX,DI
0006 03 C6                      ADD  AX,SI
0008 CF                         IRET

0009                    INTS    ENDP
```

## Interrupt Control

Although this section does not explain hardware interrupts, we introduce two instructions that control the INTR pin. The set interrupt flag instruction (STI) places a 1 into I, which enables the INTR pin. The clear interrupt flag instruction (CLI) places a 0 into I, which disables the INTR pin. The STI instruction enables INTR and the CLI instruction disables INTR. In a software interrupt service procedure we usually enable the hardware interrupts as one of the first steps. This is accomplished by the STI instruction.

## Interrupts in the Personal Computer

The interrupts found in the personal computer differ somewhat from the ones presented in Table 6–3. They differ because the original personal computers were 8086/8088-based systems. This meant that they only contained Intel-specified interrupts 0–4. This design is carried forward so newer systems are compatible with the early personal computers.

Because the personal computer is operated in the real mode, the interrupt vector table is located at addresses 00000H–003FFH. The assignments used by the computer system are listed in Table 6–4. Notice that these differ somewhat from the assignments in Table 6–3. Some of the interrupts shown in this table are used in example programs in later chapters. An example is the clock tick, which is extremely useful for timing events because it occurs a constant 18.2 times a second in all personal computers.

Interrupts 00H–1FH and 70H–77H are present in the computer no matter what operating system is installed. If DOS is installed, interrupts 20H–2FH are also present.

**TABLE 6–4** Interrupt assignments for the personal computer

| *Number* | *Function* |
|---|---|
| 0 | Divide error |
| 1 | Single step (debug) |
| 2 | Nonmaskable interrupt pin |
| 3 | Breakpoint |
| 4 | Arithmetic overflow |
| 5 | Print screen key and BOUND instruction |
| 6 | Illegal instruction error |
| 7 | Coprocessor not present interrupt |
| 8 | Clock tick (hardware) (Approximately 18.2 Hz) |
| 9 | Keyboard (hardware) |
| A | Hardware interrupt 2 (system bus) (cascade in AT) |
| B–F | Hardware interrupts 3–7 (system bus) |
| 10 | Video BIOS |
| 11 | Equipment environment |
| 12 | Conventional memory size |
| 13 | Direct disk service |
| 14 | Serial COM port service |
| 15 | Miscellaneous service |
| 16 | Keyboard service |
| 17 | Parallel port LPT service |
| 18 | ROM BASIC |
| 19 | Reboot |
| 1A | Clock service |
| 1B | Control-break handler |
| 1C | User timer service |
| 1D | Pointer for video parameter table |
| 1E | Pointer for disk drive parameter table |
| 1F | Pointer for graphics character pattern table |
| 20 | Terminate program |
| 21 | DOS services |
| 22 | Program termination handler |
| 23 | Control C handler |
| 24 | Critical error handler |
| 25 | Read disk |
| 26 | Write disk |
| 27 | Terminate and stay resident |
| 28 | DOS idle |
| 2F | Multiplex handler |
| 70–77 | Hardware interrupts 8–15 (AT style computer) |

## 6–4 MACHINE CONTROL AND MISCELLANEOUS INSTRUCTIONS

The last category of real-mode instructions found in the 8086–80486/Pentium microprocessor is the machine control and miscellaneous group. These instructions provide

control of the carry bit, sample the $\overline{\text{BUSY/TEST}}$ pin, and perform various other functions. Because many of these instructions are used in hardware control, they need only be explained briefly at this point. We cover most of these instructions in more detail in later chapters that deal with the hardware and programs that control the hardware.

## Controlling the Carry Flag Bit

The carry flag (C) propagates the carry or borrow in multiple-word/doubleword addition and subtraction. It also indicates errors in procedures. There are three instructions that control the contents of the carry flag: STC *(set carry)*, CLC *(clear carry)*, and CMC *(complement carry)*.

Because the carry flag is seldom used, except with multiple-word addition and subtraction, it is available for other uses. The most common task for the carry flag is to indicate error upon return from a procedure. Suppose that a procedure reads data from a disk memory file. This operation can be successful or an error can occur, such as file-not-found. Upon return from this procedure, if C = 1, an error has occurred and if C = 0, no error occurred. Most of the DOS and BIOS procedures use the carry flag to indicate error conditions.

## WAIT

The WAIT instruction monitors the hardware $\overline{\text{BUSY}}$ pin on the 80286–80386 and the $\overline{\text{TEST}}$ pin on the 8086/8088. The name of this pin was changed in the 80286 microprocessor from $\overline{\text{TEST}}$ to $\overline{\text{BUSY}}$. If the WAIT instruction executes while the $\overline{\text{BUSY}}$ pin = 0, nothing happens and the next instruction executes. If the $\overline{\text{BUSY}}$ pin = 1 when the WAIT instruction executes, the microprocessor waits for the $\overline{\text{BUSY}}$ pin to return to a logic 0.

The $\overline{\text{BUSY}}$ pin of the 80286–80386 is usually connected to the $\overline{\text{BUSY}}$ pin of the 8087–80387 numeric coprocessor. This connection allows the 8086–80386 to wait until the coprocessor finishes a task. Because the coprocessor is inside the 80486/Pentium, the $\overline{\text{BUSY}}$ pin is not present.

## HLT

The halt instruction (HLT) stops the execution of software. There are three ways to exit a halt: by an interrupt, by a hardware reset, or during a DMA operation. This instruction normally appears in a program to wait for an interrupt. It often synchronizes external hardware interrupts with the software system.

## NOP

When the microprocessor encounters a no-operation instruction (NOP), it takes a short time to execute. A NOP performs absolutely no operation and often pads software with space for future machine language instructions. If you are developing machine language programs, which is *extremely rare*, we recommend that you place about 10 NOPs in your program at 50-byte intervals, in case you need to add instructions at some future point. A NOP also finds application in time delay programs to waste short periods of time.

## LOCK Prefix

The LOCK prefix appends an instruction and causes the $\overline{\text{LOCK}}$ pin to become a logic 0. The $\overline{\text{LOCK}}$ pin often disables external bus masters or other system components. The LOCK prefix causes the lock pin to activate for the duration of a locked instruction. If we lock more than one sequential instruction, the $\overline{\text{LOCK}}$ pin remains a logic 0 for the duration of the sequence of locked instructions. The LOCK:MOV AL,[SI] instruction is an example of a locked instruction.

## ESC

The escape (ESC) instruction passes information to the 8087–80387 numeric coprocessor. Whenever an ESC instruction executes, the microprocessor provides the memory address if required, but otherwise performs a NOP. The 8087–80387 uses six bits of the ESC instruction to obtain its opcode and begin executing a coprocessor instruction.

The ESC opcode never appears in a program as ESC. In its place are a set of coprocessor instructions (FLD, FST, FMUL, etc.) that assemble as ESC instructions for the coprocessor. We provide more detail in the chapter that details the 8087–80387 coprocessor.

## BOUND

The BOUND instruction is a compare instruction that can cause an interrupt (vector type number 5). This instruction compares the contents of any 16-bit or 32-bit register against the contents of two words or doublewords of memory: an upper and a lower boundary. If the value in the register compared with memory is *not* within the upper and lower boundary, a type 5 interrupt ensues. If it is within the boundary, the next instruction in the program executes.

For example, if the BOUND SI,DATA instruction executes, word-sized location DATA contains the lower boundary and word-sized location DATA +2 bytes contains the upper boundary. If the number contained in SI is less than memory location DATA or greater than memory location DATA +2 bytes, a type 5 interrupt occurs. Note that when this interrupt occurs, the return address points to the BOUND instruction, not the instruction following BOUND. This differs from a normal interrupt where the return address points to the next instruction in the program.

## ENTER and LEAVE

The ENTER and LEAVE instructions are used with stack frames. A stack frame is a mechanism used to pass parameters to a procedure through the stack memory. The stack frame also holds local memory variables for the procedure. Stack frames provide dynamic areas of memory for procedures in multiuser environments.

The ENTER instruction creates a stack frame by pushing BP onto the stack and then loading BP with the uppermost address of the stack frame. This allows stack frame variables to be accessed through the BP register. The ENTER instruction contains two operands: the first operand specifies the number of bytes to reserve for variables on the stack frame, and the second specifies the level of the procedure.

Suppose that an ENTER 8,0 instruction executes. This instruction reserves eight bytes of memory for the stack frame and the zero specifies level 0. Figure 6–9 shows the stack frame set up by this instruction. Note that this instruction stores BP onto the top of the stack. It then subtracts 8 from the stack pointer leaving eight bytes of memory space for temporary data storage. The uppermost location of this 8-byte temporary storage area is addressed by BP. The LEAVE instruction reverses this process by reloading both SP and BP with their prior values.

**EXAMPLE 6–13**

```
                        ;A sequence used to call system software that
                        ;uses parameters stored in a stack frame
                        ;
0000 C8 0004 00                 ENTER 4,0               ;create 4 byte frame

0004 A1 00C8 R                  MOV  AX,DATA1
0007 89 46 FC                   MOV  [BP-4],AX          ;save para 1
000A A1 00CA R                  MOV  AX,DATA2
000D 89 46 FE                   MOV  [BP-2],AX          ;save para 2

0010 E8 0100 R                  CALL SYS                ;call subroutine

0013 8B 46 FC                   MOV  AX,[BP-4]          ;get result 1
0016 A3 00C8 R                  MOV  DATA1,AX           ;save result 1
0019 8B 46 FE                   MOV  AX,[BP-2]          ;get result 2
001C A3 00CA R                  MOV  DATA2,AX           ;save result 2

001F C9                         LEAVE
                                  .     .
                                  .     .
                                (other software continues here)
                                  .     .
                                  .     .
                        ;system subroutine that uses the stack frame parameters
```

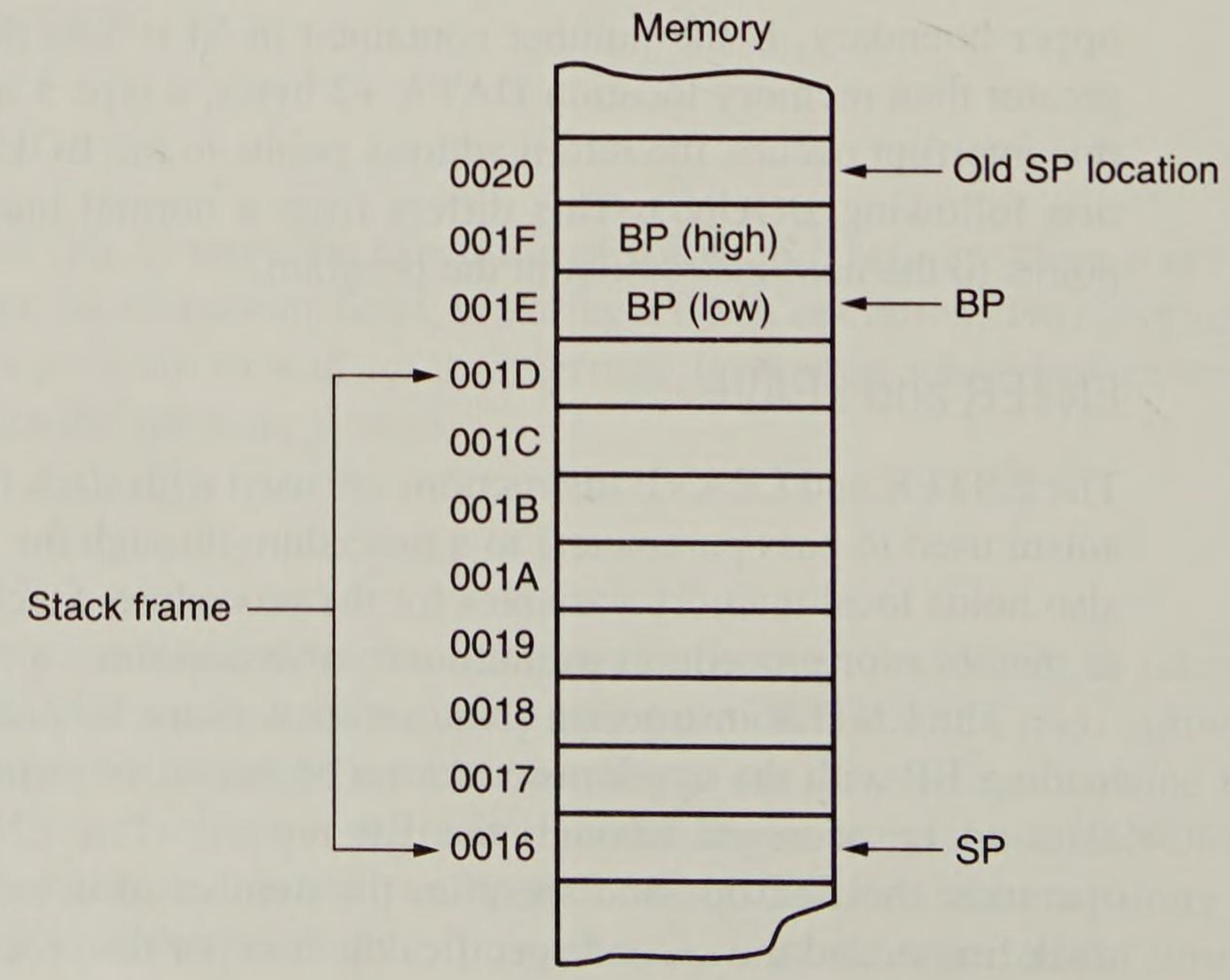



**FIGURE 6–9** The stack frame created by the ENTER 8,0 instruction. Notice that BP is stored beginning at the top of the stack frame. This is followed by an 8-byte area called a stack frame.

```
                        ;
0100                    SYS     PROC NEAR

0100 60                         PUSHA

0101 8B 46 FC                   MOV  AX,[BP-4]        ;get para 1
0104 8B 5E FE                   MOV  BX,[BP-2]        ;get para 2
                                 .    .
                                 .    .
                                (software that uses the parameters)
                                 .    .
0130 89 46 FC                   MOV  [BP-4],AX        ;save result 1
0133 89 5E FE                   MOV  [BP-2],BX        ;save result 2

0136 61                         POPA
0137 C3                         RET

0138                    SYS     ENDP
```

Example 6–13 shows how the ENTER instruction creates a stack frame so two 16-bit parameters are passed to a system level procedure. Notice how the ENTER and LEAVE instructions appear in this program, and how the parameters pass through the stack frame to and from the procedure. This procedure uses two parameters that pass to it, and returns two results through the stack frame.

## 6–5 SUMMARY

1. There are three types of unconditional jump instructions: short, near, and far. The short jump allows a branch to within +127 and –128 bytes. The near jump (using a displacement of ±32K) allows a jump to anywhere in the current code segment (intrasegment). The far jump allows a jump to any location in the memory system (intersegment). The near jump in an 80386/80486/Pentium microprocessor is within ±2G bytes because these microprocessors can use a 32-bit signed displacement.
2. Whenever a label appears with a JMP instruction or conditional jump, the label, located in the label field, must be followed by a colon (LABEL:). The JMP DOGGY instruction jumps to memory location DOGGY:.
3. The displacement that follows a short or near jump is the distance from the next instruction to the jump location.
4. Indirect jumps are available in two forms: (1) jump to the location stored in a register and (2) jump to the location stored in a memory word (near indirect) or doubleword (far indirect).
5. Conditional jumps are all short jumps that test one or more of the flag bits: C, Z, O, P, or S. If the condition is true, a jump occurs, and if the condition is false, the next sequential instruction executes. Note that the 80386, 80486, and Pentium allow a 16-bit signed displacement for the conditional jump instructions.
6. A special conditional jump instruction (LOOP) decrements CX and jumps to the label when CX is not 0. Other forms of loop include: LOOPE, LOOPNE, LOOPZ, and

LOOPNZ. The LOOPE instruction jumps if CX is not 0, and if an equal condition exists. In the 80386/80486/Pentium microprocessor the LOOPD, LOOPED, and LOOPNED instructions also use the ECX register as a counter.

7. In the 80386/80486/Pentium microprocessor, a group of set-according-to conditions exists that either set a byte to 01H or clear it to 00H. If the condition under test is true, the operand byte is set to a 01H, and if the condition under test is false, the operand byte is cleared to 00H.
8. Procedures are groups of instructions that perform one task and are used from any point in a program. The CALL instruction links to a procedure and the RET instruction returns from a procedure. In assembly language, the PROC directive defines the name and type of procedure. The ENDP directive declares the end of the procedure.
9. The CALL instruction is a combination of a PUSH and a JMP instruction. When CALL executes, it pushes the return address on the stack and then jumps to the procedure. A near CALL places the contents of IP on the stack and a far CALL places both IP and CS on the stack.
10. The RET instruction returns from a procedure by removing the return address from the stack and placing it into IP (near return) or IP and CS (far return).
11. Interrupts are either software instructions similar to CALL or hardware signals used to call procedures. This process interrupts the current program and calls a procedure. After the procedure, a special IRET instruction returns control to the interrupted software.
12. Real-mode interrupt vectors that contain the address (IP and CS) of the interrupt service procedure are four bytes in length. The microprocessor contains 256 interrupt vectors in the first 1K byte of memory. The first 32 are defined by Intel, the remaining 224 are user interrupts. In protected-mode operation, the interrupt vector is eight bytes in length, and the interrupt vector table may be relocated to any section of the memory system.
13. Whenever an interrupt is accepted by the microprocessor, the flags, IP, and CS are pushed on the stack. Besides pushing the flags, the T and I flag bits are cleared to disable both the trace function and the INTR pin. The final event that occurs for the interrupt is that the interrupt vector is fetched from the vector table and a jump to the interrupt service procedure occurs.
14. Software interrupt instructions (INT) often replace system calls. Software interrupts save three bytes of memory each time they replace CALL instructions.
15. A special return instruction (IRET) must be used to return from an interrupt service procedure. The IRET instruction not only removes IP and CS from the stack, it also removes the flags from the stack.
16. Interrupt on an overflow (INTO) is a conditional interrupt that calls an interrupt service procedure if the overflow flag (O) = 1.
17. The interrupt enable flag (I) controls the INTR pin connection on the microprocessor. If the STI instruction executes, it sets I to enable the INTR pin. If the CLI instruction executes, it clears I to disable the INTR pin.
18. The carry flag bit (C) is cleared, set, and complemented by the CLC, STC, and CMC instructions.
19. The WAIT instruction tests the condition of the $\overline{\text{BUSY}}$ or $\overline{\text{TEST}}$ pin on the microprocessor. If $\overline{\text{BUSY}}$ or $\overline{\text{TEST}}$ = 0, WAIT does not wait, but if $\overline{\text{BUSY}}$ or $\overline{\text{TEST}}$ = 1, WAIT continues testing the $\overline{\text{BUSY}}$ or $\overline{\text{TEST}}$ pin until it becomes a logic 0. Note that

the 8086/8088 contains the $\overline{\text{TEST}}$ pin while the 80286–80386 contains the $\overline{\text{BUSY}}$ pin. The 80486 and Pentium do not contain a $\overline{\text{BUSY}}$ or $\overline{\text{TEST}}$ pin.

20. The LOCK prefix causes the $\overline{\text{LOCK}}$ pin to become a logic 0 for the duration of the locked instruction. The ESC instruction passes instruction to the numeric coprocessor.
21. The BOUND instruction compares the contents of any 16-bit register against the contents of two words of memory: an upper and a lower boundary. If the value in the register compared with memory is *not* within the upper and lower boundary, a type 5 interrupt ensues.
22. The ENTER and LEAVE instructions are used with stack frames. A stack frame is a mechanism used to pass parameters to a procedure through the stack memory. The stack frame also holds local memory variables for the procedure. The ENTER instruction creates the stack frame, and the LEAVE instruction removes the stack frame from the stack. The BP register addresses stack frame data.

## 6–6 QUESTIONS AND PROBLEMS

1. What is a short JMP?
2. Which type of JMP is used when jumping to any location within the current code segment?
3. Which JMP instruction allows the program to continue execution at any memory location in the system?
4. Which JMP instruction is five bytes long?
5. What is the range of a near jump in the 80386/80486/Pentium microprocessor?
6. Which type of JMP instruction (short, near, or far) assembles for the following:
   (a) if the distance is 0210H bytes
   (b) if the distance is 0020H bytes
   (c) if the distance is 10000H bytes
7. What can be said about a label that is followed by a colon?
8. The near jump modifies the program address by changing which register or registers?
9. The far jump modifies the program address by changing which register or registers?
10. Explain what the JMP AX instruction accomplishes. Also identify it as a near or a far jump instruction.
11. Contrast the operation of a JMP DI with a JMP [DI].
12. Contrast the operation of a JMP [DI] with a JMP FAR PTR [DI].
13. List the five flag bits tested by the conditional jump instructions.
14. Describe how the JA instruction operates.
15. When will the JO instruction jump?
16. Which conditional jump instructions follow the comparison of signed numbers?
17. Which conditional jump instructions follow the comparison of unsigned numbers?
18. Which conditional jump instructions test both the Z and C flag bits?
19. When does the JCXZ instruction jump?
20. Which SET instruction is used to set AL if the flag bits indicate a zero condition?
21. The 8086 LOOP instruction decrements register _____ and tests it for a 0 to decide if a jump occurs.

22. The 80486 LOOPD instruction decrements register _____ and tests it for a 0 to decide if a jump occurs.
23. Explain how the LOOPE instruction operates.
24. Develop a short sequence of instructions that stores a 00H into 150H bytes of memory beginning at extra-segment memory location DATA. You must use the LOOP instruction to help perform this task.
25. Develop a sequence of instructions that searches through a block of 100H bytes of memory. This program must count all the unsigned numbers that are above 42H and all that are below 42H. Byte-sized data-segment memory location UP must contain the count of numbers above 42H and data-segment memory location DOWN must contain the count of numbers below 42H.
26. What is a procedure?
27. Explain how the near and far CALL instructions function.
28. How does the near RET instruction function?
29. The last executable instruction in a procedure must be a _____.
30. Which directive identifies the start of a procedure?
31. How is a procedure identified as near or far?
32. Explain what the RET 6 instruction accomplishes.
33. Write a near procedure that cubes the contents of the CX register. This procedure may not affect any register except CX.
34. Write a procedure that multiplies DI by SI and then divides the result by 100H. Make sure that the result is left in AX upon returning from the procedure. This procedure may not change any register except AX.
35. Write a procedure that sums EAX, EBX, ECX, and EDX. If a carry occurs, place a logic 1 in EDI. If no carry occurs, place a 0 in EDI. The sum should be found in EAX after the execution of your procedure.
36. What is an interrupt?
37. Which software instructions call an interrupt service procedure?
38. How many different interrupt types are available in the microprocessor?
39. What is the purpose of interrupt vector type number 0?
40. Illustrate the contents of an interrupt vector and explain the purpose of each part.
41. How does the IRET instruction differ from the RET instruction?
42. What is the IRETD instruction?
43. The INTO instruction only interrupts the program for what condition?
44. The interrupt vector for an INT 40H instruction is stored at what memory locations?
45. What instructions control the function of the INTR pin?
46. Which personal computer interrupt services the parallel LPT port?
47. Which personal computer interrupt services the keyboard?
48. What instruction tests the $\overline{\text{BUSY}}$ pin?
49. When will the BOUND instruction interrupt a program?
50. An ENTER 16,0 instruction creates a stack frame that contains _____ bytes.
51. Which register moves to the stack when an ENTER instruction executes?
52. Which instruction passes opcodes to the numeric coprocessor?

# CHAPTER 7

# Programming with DOS and BIOS Functions

## INTRODUCTION

This chapter develops programs and programming techniques using the Microsoft MASM macro assembler, the DOS function calls, and both the system and video BIOS function calls. Some of the DOS and BIOS function calls are used in this chapter, but all are explained in complete detail in Appendix A. Please review the function calls as you read this chapter. The MASM assembler has already been explained and demonstrated, but there are many more features to learn.

Some programming techniques explained in this chapter include: disk read and writes, macro sequences, keyboard and display manipulation, program modules, library files, and other important programming techniques. This chapter, although meant as an introduction to programming, provides valuable programming techniques that are important for efficient program development with the personal computer, using either PCDOS* or MSDOS† as a springboard.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Use the MASM assembler and linker program to create programs that contain more than one module.
2. Explain the use of EXTRN and PUBLIC as they apply to modular programming.
3. Set up a library file that contains commonly used subroutines.
4. Write and use MACRO and ENDM to develop macro sequences used with linear programming.

---

*PCDOS is a registered trademark of IBM Corporation.

†MSDOS is a registered trademark of Microsoft Corporation.

5. Develop programs using DOS function calls.
6. Differentiate a DOS function call from a BIOS function call.
7. Use modes 12H and 13H of the VGA display system.
8. Using DOS function calls, read and write to the disk memory system.
9. Differentiate sequential and random access files.

## 7–1 MODULAR PROGRAMMING

Most programs are too large to be developed by a single person. Rather, programs are usually developed by teams of programmers. The linker is provided with MSDOS or PCDOS so programming modules can be linked or joined together to form a complete program. This section describes the linker, the linking task, library files, EXTRN, and PUBLIC as they apply to program modules and modular programming.

### The Assembler and Linker

The *assembler program* converts a symbolic source module or file (FILENAME.ASM) into a hexadecimal object file (FILENAME.OBJ). We have seen many examples of symbolic source files in prior chapters. Example 7–1 shows the dialog (using MASM 5.10) that appears as a source module named FILE.ASM is assembled. Whenever you create a source file it must have an extension of ASM. Note that the extension is not typed into the assembler prompt when assembling a file. Source files are created using the editor that comes with the assembler or by almost any other editor or word processor that is capable of generating an ASCII file.

**EXAMPLE 7–1**

```
A>MASM

Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1989.  All rights reserved.

Source filename [.ASM]: FILE
Object filename [FILE.OBJ]: FILE
Source listing [NUL.LST]: FILE
Cross reference [NUL.CRF]: FILE
```

The assembler program (MASM) asks for the source file name, the object file name, the list file name, and a cross-reference file name. In most cases the name for each of these will be the same as the source file. The object file (.OBJ) is not executable, but is designed as an input file to the linker. The source listing file (.LST) contains the assembled version of the source file and its hexadecimal machine language equivalent. The cross-reference file (.CRF) lists all labels and pertinent information required for cross-referencing.

The *linker program* reads the object files created by the assembler program, and links them together into a single execution file. An *execution file* is created with the filename extension EXE. Execution files are executed by typing the file name at the DOS

prompt (A>). An example execution file is FROG.EXE, executed by typing FROG at the DOS command prompt.

If a file is short enough, less than 64K bytes in length, it can be converted from an execution file to a *command file* (.COM). The command file is slightly different from an execution file in that the program must be originated at location 100H before it can execute. The program EXE2BIN is used for converting an execution file into a command file. For example, EXE2BIN PROG.EXE PROG.COM will convert PROG.EXE into the command file PROG.COM. The main advantage of a command file is that is loads from the disk into the computer much more quickly than an execution file. It also requires slightly less disk storage space than the execution file. If MASM version 6.X is in use, the EXE2BIN program is not needed. The WORKBENCH editor program for MASM version 6.X creates the command file directly if the command file (.COM) option is selected. WORKBENCH also builds the file by assembling multiple modules, if needed, and linking them.

**EXAMPLE 7–2**

```
A>LINK

Microsoft (R) Overlay Linker Version 3.64
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object Modules [.OBJ]: FROG+WHAT+DONUT
Run File [FROG.EXE]: FROG
List File [NUL.MAP]: FROG
Libraries [.LIB]: LIBS
```

Example 7–2 shows the protocol involved to use the linker program to link the files FROG, WHAT, and DONUT. The linker also links the library file (LIBS) so the procedures located within LIBS can be used with the linked execution file. To invoke the linker, type LINK at the DOS command prompt as illustrated in Example 7–2. Note that before files can be linked, they must first be assembled and they must be *error free*.

In this example, after typing LINK, the linker program asks for the Object Modules, which are created by the assembler. In this example, we have three object modules: FROG, WHAT, and DONUT. If more than one object file exists, the main program file (FROG in this example) is typed first followed by any other supporting modules. (We must use a plus sign to separate module names.)

After the program module names are typed, the linker suggests that the execution (*run-time*) file name is FROG.EXE. This may be selected by typing the same name or if desired by typing the enter key. It may also be changed to any other name at this point.

The list file is where a map of the program segments appears as created by the linking. This map contains the segment names and lengths. If the enter key is typed, no list file is created, but if a name is typed, the list file appears on the disk.

Library files are entered in the last line. In this example we entered library file name LIBS. This library contains procedures used by the other program modules.

If you are using MASM version 6.X, the programmer's WORKBENCH (PWB.EXE) program is used to edit, assemble, and link program modules. In this newer, more efficient programming medium, the DOS prompts disappear and are replaced with

pull-down menus and windows that automate the process. To create an execute or command file, we merely adjust the parameters of PWB by using the option menu, then create and edit the program. Once the program appears to be correct, we build the execute or command file by using the BUILD option in the run menu. The PWB program also allows access to the debugging tool, CODEVIEW, once the program is assembled and linked correctly.

## PUBLIC and EXTRN

The PUBLIC and EXTRN directives are very important to modular programming. We use PUBLIC to declare that labels of code, data, or entire segments are available to other program modules. We use EXTRN (external) to declare that labels or segments are external to a module. Without these statements, we could not link modules together to create a program using modular programming techniques.

**EXAMPLE 7–3**

```
                        DAT1    SEGMENT PUBLIC          ;declare entire public

                                PUBLIC  DATA1           ;declare DATA1, DATA2 public
                                PUBLIC  DATA2

0000 0064[              DATA1   DB      100 DUP (?      ;global
          ??
             ]
0064 0064[              DATA2   DB      100 DUP (?)     ;global
          ??
             ]

00C8                    DAT1    ENDS

0000                    CODES   SEGMENT 'CODE'

                                ASSUME  CS:CODES,DS:DAT1

                                PUBLIC  READ            ;declare READ public

0000                    READ    PROC FAR

0000 B4 06                      MOV     AH,6            ;read keyboard
0002 B2 FF                      MOV     DL,0FFH         ;no echo
0004 CD 21                      INT     21H
0006 74 F8                      JE      READ
0008 CB                         RET

0009                    READ    ENDP

0009                    CODES   ENDS

                                END
```

The PUBLIC directive is normally placed in the opcode field of an assembly language statement to define a label as public, so it can be used by other modules. A jump address, a data address, or an entire segment can be made public. Example 7–3 shows the PUBLIC statement used to define some labels public to other modules. When segments are made public they are combined with other public segments that contain data with the same segment name.

**EXAMPLE 7–4**

```
0000                    DAT1    SEGMENT  PUBLIC      ;declare segment public

                                EXTRN  DATA1:BYTE
                                EXTRN  DATA2:BYTE
                                EXTRN  DATA3:WORD
                                EXTRN  DATA4:DWORD

0000                    DAT1    ENDS

0000                    CODES   SEGMENT 'CODE'

                                ASSUME CS:CODES,ES:DAT1

                                EXTRN  READ:FAR

0000                    MAIN    PROC FAR

0000 B8 ---- R                  MOV  AX,DAT1
0003 8E C0                      MOV  ES,AX

0005 BF 0000 E                  MOV  DI,OFFSET DATA1
0008 B9 000A                    MOV  CX,10
000B                    MAIN1:
000B 9A 0000 ---- E             CALL READ
0010 AA                         STOSB
0011 E2 F8                      LOOP MAIN1
0013 CB                         RET

0014                    MAIN    ENDP

0014                    CODES   ENDS

                                END  MAIN
```

The EXTRN statement appears in both data and code segments to define labels as external to the segment. If data is defined as external, its size must be represented as BYTE, WORD, or DWORD. If a jump or call address is external, it must be represented as NEAR or FAR. Example 7–4 shows how the external statement is used to indicate that several labels are external to the program listed. Notice in this example that any external address or data is defined with the letter E in the hexadecimal-assembled listing.

## Libraries

Library files are collections of procedures that are used by many different programs. These procedures are assembled and compiled into a library file by the LIB program that accompanies the MASM assembler program. Libraries allow common procedures to be collected into one place so they can be used by many different applications. The library file (FILENAME.LIB) is invoked when a program is linked with the linker program.

Why bother with library files? A library file is a good place to store a collection of related procedures. When the library file is linked with a program, only the procedures required by the program are removed from the library file and added to the program. If any amount of assembly language programming is to be accomplished efficiently, a good set of library files is essential. If you are currently using C language you are already familiar with library files.

*Creating a Library File.* A library file is created with the LIB command typed at the DOS prompt. A library file is a collection of assembled .OBJ files that each performs one procedure. Example 7–5 shows two separate files (READ_KEY and ECHO) that will be used to structure a library file. Please notice that the name of the procedure must be declared PUBLIC in a library file and does not necessarily need to match the file name, although it does in this example.

**EXAMPLE 7–5**

```
                        ;The first library module is called READ_KEY.
                        ;This procedure reads a key from the keyboard
                        ;and returns with the ASCII character in AL.
                        ;
0000                    LIB     SEGMENT 'CODE'

                                ASSUME  CS:LIB

                                PUBLIC  READ_KEY

0000                    READ_KEY  PROC  FAR

0000 52                         PUSH DX
0001                    READ_KEY1:
0001 B4 06                      MOV  AH,6
0003 B2 0F                      MOV  DL,0FH
0005 CD 21                      INT  21H
0007 74 F8                      JE   READ_KEY1
0009 5A                         POP  DX
000A CB                         RET
000B                    READ_KEY  ENDP

000B                    LIB     ENDS

                                END
                        ;
                        ;This second library module is called ECHO
                        ;This procedure displays the ASCII character
                        ;in AL on the CRT screen.
                        ;
0000                    LIB     SEGMENT 'CODE'

                                ASSUME  CS:LIB

                                PUBLIC  ECHO

0000                    ECHO    PROC FAR
0000 52                         PUSH DX
0001 B4 06                      MOV  AH,6
0003 8A D0                      MOV  DL,AL
0005 CD 21                      INT  21H
0007 5A                         POP  DX
0008 CB                         RET
0009                    ECHO    ENDP

0009                    LIB     ENDS

                                END
```

After each file is assembled, the LIB program is used to combine them into a library file. The LIB program prompts for information as illustrated in Example 7–6 where these files are combined to form the library IO.

**EXAMPLE 7–6**

```
A>LIB

Microsoft (R) Library Manager Version 3.10
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Library name:IO
Library file does not exist.  Create? Y
Operations:READ_KEY+ECHO
List file:IO
```

The LIB program begins with the copyright message from Microsoft, followed by the prompt *Library name:*. The library name chosen is IO for the IO.LIB file. Because this is a new file, the library program asks if we wish to create the library file. The *Operations:* prompt is where the library module names are typed. In this example we create a library using two procedure files (READ_KEY and ECHO). The list file shows the contents of the library and is illustrated for IO.LIB in Example 7–7. The list file shows the size and names of the files used to create the library and also the public label (procedure name) that is used in the library file.

**EXAMPLE 7–7**

```
ECHO.............ECHO                READ_KEY.........READ_KEY

READ_KEY          Offset: 00000010H  Code and data size: BH
  READ_KEY

ECHO              Offset: 00000070H  Code and data size: 9H
   ECHO
```

If you must add additional library modules at a later time, type the name of the library file after invoking LIB. At the *Operations:* prompt type the new module name preceded by a *plus sign* to add a new procedure. If you must delete a library module, use a *minus sign* before the operation file name.

Once the library file is linked to your program file, only the library procedures actually used by your program are placed in the execution file. Don't forget to use the label EXTRN when specifying library calls from your program module.

## Macros

A macro is a group of instructions that normally performs one task, just as a procedure performs one task. The difference is that a procedure is accessed via a CALL instruction, while a macro is inserted in the program as a new opcode representing a sequence of instructions. A *macro* is a new opcode that you create. Macro sequences execute faster than procedures because there are no CALL and RET instructions to execute. The instructions of the macro

sequence are placed in your program at the point where they are invoked. When software is developed using macro sequences it flows from the top down.

Macro sequences are ideal for systems that contain cache memory or are required to execute software with maximum speed and efficiency. Programs that use macro sequences in place of procedures are often called *linear programs* because they flow from the top to the bottom without any calls to procedures. If a cache memory is used in the computer system, it is often desirable to develop programs that use macro sequences, because the entire program can often be executed from the cache. This significantly increases its execution speed. If a procedure is called, the cache must often be reloaded, requiring additional time. If a macro is invoked, no additional time is needed because no jump to the procedure occurs. Cache memory looks ahead and often loads sequential instructions before they are executed by the microprocessor. It does this by loading the cache with the next sequential section of four 32-bit doublewords. If a procedure is called, it is not normally in the next section of memory, which requires another section of the memory to be loaded to the cache. This requires additional time on the part of the microprocessor's bus interface unit.

The MACRO and ENDM directives are used to delineate a macro sequence. The first statement of a macro is the MACRO statement. The MACRO statement contains the name of the macro and any parameters associated with it. An example is MOVE MACRO A,B that defines the macro name as MOVE. This new *opcode* (MOVE) uses two parameters, A and B. The last statement of a macro is the ENDM instruction, on a line by itself without a label.

**EXAMPLE 7–8**

```
                        MOVE    MACRO  A,B          ;;moves word from B to A
                                PUSH AX
                                MOV  AX,B
                                MOV  A,AX
                                POP  AX
                                ENDM

                        MOVE    VAR1,VAR2          ;use macro MOVE

0000 50                 1       PUSH AX
0001 A1 0002 R          1       MOV  AX,VAR2
0004 A3 0000 R          1       MOV  VAR1,AX
0007 58                 1       POP  AX

                        MOVE    VAR3,VAR4          ;use macro MOVE

0008 50                 1       PUSH AX
0009 A1 0006 R          1       MOV  AX,VAR4
000C A3 0004 R          1       MOV  VAR3,AX
000F 58                 1       POP  AX
```

Example 7–8 shows how a macro is created and used in a program. This macro moves the word-sized contents of memory location B into word-sized memory location A. After the macro is defined in the example, it is used twice. The macro is *expanded* in this example so you can see how it assembles to generate the moves. If a hexadecimal machine language statement is followed by a 1 or other number, it is a macro expansion statement. If a macro uses a second macro (nesting), the statement is followed by a 2, and so forth.

The expansion statements were not typed in the source program. The assembler inserted them each time the macro was invoked by the program. Notice that the comment in the macro is preceded with a ;; instead of ; as is customary.

*Local Variable in a Macro.* Sometimes macros contain local variables. A *local variable* is one that appears in the macro, but is not available outside the macro. To define a local variable, we use the LOCAL directive. Example 7–9 shows how a local variable, used as a jump address, appears in a macro definition. If this jump address is not defined as a local label, the assembler will flag it with a duplicate label error on subsequent attempts to use the macro in a program.

**EXAMPLE 7–9**

```
                        READ    MACRO   A               ;;reads keyboard
                                LOCAL   READ1           ;;define READ1 as local
                                PUSH    DX
                        READ1:
                                MOV   AH,6
                                MOV   DL,0FFH
                                INT   21H
                                JE    READ1
                                MOV   A,AL
                                POP   DX
                                ENDM

                                READ VAR5               ;read key

0000 52                 1               PUSH DX
0001                    1       ??0000:
0001 B4 06              1               MOV   AH,6
0003 B2 FF              1               MOV   DL,0FFH
0005 CD 21              1               INT   21H
0007 74 F8              1               JE    ??0000
0009 A2 0008 R          1               MOV   VAR5,AL
000C 5A                 1               POP   DX

                                READ VAR6               ;read key

000D 52                 1               PUSH DX
000E                    1       ??0001:
000E B4 06              1               MOV   AH,6
0010 B2 FF              1               MOV   DL,0FFH
0012 CD 21              1               INT   21H
0014 74 F8              1               JE    ??0001
0016 A2 0009 R          1               MOV   VAR6,AL
0019 5A                 1               POP   DX
```

This example macro reads a character from the keyboard and stores it into the byte-sized memory location indicated as a parameter with the macro. Notice how the local label READ1 is treated in the expanded macros. Local variables always appear as ??nnnn, where nnnn is a number identifying it.

The LOCAL directive must always immediately follow the MACRO directive without any intervening blank lines or comments. If a comment or blank line appears between the MACRO and LOCAL statements, the assembler indicates an error and will not accept the variable as local.

*Placing Macro Definitions in Their Own Module.* Macro definitions can be placed in the program file as shown, or can be placed in their own macro module. A file can be created that contains only macros that are to be included with other program files. We use the INCLUDE directive to indicate that a program file will include a module that contains external macro definitions. Although this is not a library file, it functions, for all practical purposes, as a library of macro sequences. A macro include file is an ASCII file generated by an editor or word processor. The include file is never assembled and linked, but is included with another program file.

When macro sequences are placed in a file (often with the extension INC or MAC) they do not contain PUBLIC statements. If a file called MACRO.MAC contains macro sequences, the include statement is placed in the program file as INCLUDE C:\ASSM\MACRO.MAC. Notice that the macro include file is on drive C, subdirectory ASSM in this example. The INCLUDE statement includes these macros just as if you had typed them into the file. No EXTRN statement is needed to access the macro statements that have been included in the program file.

## The Modular Programming Approach

The modular programming approach often involves a team of people with each assigned to a different programming task. This allows the team manager to assign various portions of the program to different team members. Often the team manager develops the system flow-chart or shell and then divides it into modules for team members.

A team member might be assigned the task of developing a macro definition file. This file might contain macro definitions that handle the I/O operations for the system. Another team member might be assigned the task of developing the procedures used for the system. In most cases, the procedures are organized as a library file linked to the program modules. Finally, several program files or modules might be used for the final system, each developed by different team members.

This approach requires good documentation and considerable communication between team members. Documentation is essential so that modules interface correctly. Communication between members plays a key role in this approach.

---

# 7–2 USING THE KEYBOARD AND VIDEO DISPLAY

Today, most programs make use of the keyboard and video display on a personal computer. This section of the text explains how to use the keyboard and video display connected to an IBM PC or compatible computer running under either MSDOS or PCDOS.

## Reading the Keyboard with DOS Functions

The keyboard in a personal computer is often read via a DOS function call. A complete listing of the DOS function calls appears in Appendix A. This section uses the INT 21H instruction with various DOS function calls to read the keyboard. Data read from the keyboard is either in ASCII-coded form or in extended ASCII-coded form. The exact form depends on which keys are typed on the keyboard.

**TABLE 7–1** Extended ASCII-coded keyboard data

| | *First* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |
| Second | | | | | | | | | |
| 0 | — | aQ | aD | aB | — | down arrow | cF3 | aF9 | a9 |
| 1 | aESC | aW | aF | aN | — | page down | cF4 | aF10 | a0 |
| 2 | — | aE | aG | aM | — | insert | cF5 | — | a- |
| 3 | c2 | aR | aH | a, | — | delete | cF6 | — | a= |
| 4 | — | aT | aJ | a. | — | sF1 | cF7 | — | — |
| 5 | — | aY | aK | a/ | — | sF2 | cF8 | — | F11 |
| 6 | — | aU | aL | — | — | sF3 | cF9 | — | F12 |
| 7 | — | aI | a; | a* | home | sF4 | cF10 | — | sF11 |
| 8 | — | aO | a' | — | up arrow | sF5 | aF1 | a1 | sF12 |
| 9 | — | aP | a` | — | page up | sF6 | aF2 | a2 | cF11 |
| A | — | a[ | — | a- | — | sF7 | aF3 | a3 | cF12 |
| B | — | a] | a\ | — | left arrow | sF8 | aF4 | a4 | aF11 |
| C | — | aENT | aZ | — | — | sF9 | aF5 | a5 | aF12 |
| D | — | — | aX | — | right arrow | sF10 | aF6 | a6 | — |
| E | aBS | aA | aC | — | a+ | cF1 | aF7 | a7 | — |
| F | sTAB | aS | — | — | end key | cF2 | aF8 | a8 | — |

*Note*: a = alternate key, c = control key, and s = shift key.

The ASCII-coded data appear as outlined in Chapter 1. Notice that these codes correspond to most of the keys on the keyboard. The extended ASCII codes apply to the printer or video screen. Also available through the keyboard is a different set of extended ASCII-coded *keyboard* data. Table 7–1 lists most of the extended ASCII codes obtained with various keys and key combinations. These are different from the extended ASCII printer/video display codes in Chapter 1. Notice that most keys on the keyboard have alternative key codes. The function keys have four sets of codes selected by the function keys, the shift function keys, the alternate function keys, and the control function keys.

There are three ways to read the keyboard. The first method reads a key and echoes (or displays) the key on the video screen. A second way tests to see whether a key is pressed; if it is pressed, it reads the key, otherwise it immediately returns without any key. The third way allows an entire character line to be read from the keyboard.

*Reading a Key with an Echo.* Example 7–10 shows how a key is read from the keyboard and *echoed* (sent) back out to the video display. Although this is the easiest way to read a key, it is also the most limited because it always echoes (displays) the character to the screen even if it is an unwanted character. The DOS function number 01H also responds to the control C key and exits the program to DOS if it is typed.

**EXAMPLE 7–10**

```
0000                    KEY     PROC FAR

0000 B4 01                      MOV  AH,1               ;function 01H
```

```
0002 CD 21                      INT   21H             ;read key
0004 0A C0                      OR    AL,AL           ;test for 00H, clear carry
0006 75 03                      JNZ   KEY1
0008 CD 21                      INT   21H             ;get extended
000A F9                         STC                   ;indicate extended
000B                    KEY1:
000B CB                         RET

000C                    KEY     ENDP
```

To read and echo a character, the AH register is loaded with DOS function number 01H. This is followed by the INT 21H instruction. All DOS functions use AH to hold the function number before the INT 21H DOS function call. Also note that the DOS INT 21H instructions generally save all registers except the contents of the accumulator, which may change for certain functions. Upon return from the INT 21H, the AL register contains the ASCII character typed and the video display shows the typed character. The return from this DOS function call does not occur until a key is actually typed. If AL = 0 after the return, the INT 21H instruction must again be executed to obtain the extended ASCII-coded character. The procedure in Example 7–10 returns with carry set (1) to indicate an extended ASCII character and carry cleared (0) to indicate a normal ASCII character.

*Reading a Key with No Echo.* The best single-character-key reading function is function number 06H. This function reads a key without an echo to the screen. It also returns with extended ASCII characters and *does not* respond to the control C key. This function uses AH for the function number (06H) and DL = 0FFH to indicate that the function call (INT 21H) will read the keyboard without an echo.

**EXAMPLE 7–11**

```
0000                    KEYS    PROC FAR

0000 B4 06                      MOV   AH,6            ;function 06H
0002 B2 FF                      MOV   DL,0FFH
0004 CD 21                      INT   21H             ;read key
0006 74 F8                      JE    KEYS            ;if no key
0008 0A C0                      OR    AL,AL           ;test for 00H, clear carry
000A 75 03                      JNE   KEYS1
000C CD 21                      INT   21H             ;get extended
000E F9                         STC                   ;indicate extended
000F                    KEYS1:
000F CB                         RET

0010                    KEYS    ENDP
```

Example 7–11 shows a procedure that uses function number 06H to read the keyboard. This procedure performs as did Example 7–10, except that no character is echoed to the video display. You must display the character at some other point, or nothing that is typed will be visible.

If you examine this procedure, there is one other difference from the procedure in Example 7–10. Function call number 06H returns from the INT 21H even if no key is typed, while function call 01H waits for a key to be typed. This is an important difference that should be noted. This feature allows software to perform other tasks between check-

ing the keyboard for a character. For example, we could display the time on the video screen or increment a random number while waiting for a keystroke.

*Reading an Entire Line with Echo.* It is sometimes advantageous to read an entire line of data with one function call. Function call number 0AH reads an entire line of information—up to 255 characters—from the keyboard. It continues to acquire keyboard data until the enter key (0DH) is typed or the count expires. This function requires that AH = 0AH and DS:DX addresses the keyboard buffer (a memory area where the ASCII keyboard data are to be stored). The first byte of the buffer area contains the maximum number of keyboard characters read by this function. If the number of characters typed exceeds this maximum number, the DOS function returns, just as if the enter key were typed. The second byte of the buffer contains the count of the actual number of characters typed, and the remaining locations in the buffer contain the ASCII keyboard data.

**EXAMPLE 7–12**

```
0000                    COD     SEGMENT 'CODE'

                                ASSUME  CS:COD,DS:COD

0000  0101[             BUF1    DB    257 DUP (?)
          ????
              ]
0102  0101[             BUF2    DB    257 DUP (?)
          ????
              ]

0204                    MAIN    PROC FAR

0204 8C C8                      MOV   AX,CS
0206 8E D8                      MOV   DS,AX

0208 BA 0000 R                  MOV   DX,OFFSET BUF1  ;address buffer 1
020B C7 06 0000 R 00FF          MOV   BUF1,255        ;maximum count
0211 E8 0221 R                  CALL  LINE            ;read first line

0214 BA 0102 R                  MOV   DX,OFFSET BUF2  ;address buffer 2
0217 C7 06 0102 R 00FF          MOV   BUF2,255        ;maximum count
021D E8 0221 R                  CALL  LINE            ;read second line

0220 CB                         RET

0221                    MAIN    ENDP

0221                    LINE    PROC NEAR

0221 B4 0A                      MOV   AH,0AH          ;function 0AH
0223 CD 21                      INT   21H
0225 C3                 RET

0226                    LINE    ENDP

0226                    COD     ENDS

                                END   MAIN
```

Example 7–12 shows how this function reads two lines of information into two memory buffers (BUF1 and BUF2). Before the call to the DOS function through procedure LINE, the first byte of the buffer is loaded with a 255 (0FFH) so up to 255 characters can be typed. If you assemble and execute this program, the first line is accepted and so is the second. The only problem is that the second line appears on top of the first line, erasing the first line. The next section of the text explains how to output characters to the video display to solve this problem and also how to display memory data on the video screen.

## Writing to the Video Display with DOS Functions

For almost any program written, data must be displayed on the video display. Video data are displayed in a number of different ways using the DOS function calls. We use function 02H or 06H to display one character at a time or function 09H to display an entire string of characters. Functions 02H and 06H are identical, but we tend to use function 06H because it can also be used to read a key.

*Displaying a Single ASCII Character.* DOS functions 02H and 06H are explained together because they are identical for displaying ASCII data. Example 7–13 shows how this function displays a carriage return (0DH) and a line feed (0AH). Here, a macro called DISP (display) displays the carriage return and line feed. The combination of a carriage return and a line feed moves the cursor to the next line at the left margin of the video screen. Double-spacing is accomplished by using the DISP macro twice. The DISP macro can correct the problem that occurred between the lines typed through the keyboard in Example 7–12 by inserting a carriage return and line feed between the lines read from the keyboard.

**EXAMPLE 7–13**

```
                        DISP    MACRO  A                ;display A
                                MOV  AH,06H
                                MOV  DL,A
                                INT  21H
                                ENDM

                        DISP    0DH                     ;carriage return

0000 B4 06          1           MOV  AH,06H
0002 B2 0D          1           MOV  DL,0DH
0004 CD 21          1           INT  21H

                        DISP    0AH                     ;line feed

0006 B4 06          1           MOV  AH,06H
0008 B2 0A          1           MOV  DL,0AH
000A CD 21          1           INT  21H
```

*Displaying a Character String.* A character string is a series of ASCII-coded characters that end with a $ (24H) when used with DOS function call number 09H. Example 7–14 shows how a message is displayed at the current cursor position on the video display. Function call number 09H requires that DS:DX address the character string before executing the INT 21H.

**EXAMPLE 7–14**

```
0000                    COD     SEGMENT 'CODE'
                                ASSUME  CS:COD,DS:COD

0000 0D 0A 0A 54 68     MES     DB    0DH,0AH,0AH,'This is a test line','$'
     69 73 20 69 73
     20 61 20 74 65
     73 74 20 6C 69
     6E 65 24

0017                    MAIN    PROC FAR

0017 8C C8                      MOV   AX,CS             ;load DS
0019 8E D8                      MOV   DS,AX

001B B4 09                      MOV   AH,09H            ;function 09H
001D BA 0000 R                  MOV   DX,OFFSET MES
0020 CD 21                      INT   21H

0022 B4 4C                      MOV   AH,4CH            ;function 4CH
0024 CD 21                      INT   21H

0026                    MAIN    ENDP

0026                    COD     ENDS

                                END   MAIN
```

This example program can be entered into the assembler, linked, and executed to produce *"This is a test line"* on the video display. Notice that an additional DOS function call is appended to the end of this program. Function call 4CH returns the system to the DOS prompt at the end of the program. We often use function number 4CH to return to DOS. We can also use function 00H, but this function has no provision for errors and is considered obsolete. The AL register contains a 00H for no error and other values that indicate errors when a program is terminated with DOS function number 4CH. A batch program (like AUTOEXEC.BAT) can use these error codes to make decisions. The IF ERRORLEVEL=1 GOTO ONE is an example batch file if the statement that tests the error code passes back to DOS by function 4CH.

*Consolidated Read Key and Echo Library Procedures.* Example 7–15 illustrates two procedures that could be assembled, linked, and added to a library file. The READ procedure reads a keyboard character and returns with either the ASCII or extended ASCII character in AL. Upon the return, if carry is set, AL contains the extended ASCII code. If carry is cleared, AL contains the standard ASCII code. The ECHO procedure displays ASCII-coded characters located in AL at the current cursor position. These two programs are the basis for keyboard and display operations in many programs.

**EXAMPLE 7–15**

```
0000                    LIB     SEGMENT 'CODE'
                                ASSUME  CS:LIB

                                PUBLIC   READ
                                PUBLIC   ECHO
```

```
                        ;
                        ;Procedure that reads a key from the keyboard (no echo)
                        ;if CF = 0, AL = standard ASCII character
                        ;if CF = 1, AL = extended ASCII character
                        ;
0000                    READ    PROC FAR

0000 52                         PUSH DX                 ;save DX
0001 B4 06                      MOV  AH,6               ;DOS function 06H
0003 B2 FF                      MOV  DL,0FFH
0005                    READ1:
0005 CD 21                      INT  21H
0007 74 FC                      JE   READ1              ;if no key
0009 0A C0                      OR   AL,AL              ;test for extended
000B 75 03                      JNZ  READ2              ;if standard ASCII

000D CD 21                      INT  21H                ;get extended
000F F9                         STC                     ;set carry
0010                    READ2:
0010 5A                         POP  DX
0011 CB                         RET

0012                    READ    ENDP
                        ;
                        ;Procedure that displays the ASCII character in AL
                        ;
0012                    ECHO    PROC FAR

0012 52                          PUSH DX
0013 B4 06                       MOV  AH,6              ;DOS function 06H
0015 8A D0                       MOV  DL,AL             ;AL to DL
0017 CD 21                       INT  21H
0019 5A                          POP  DX
001A CB                          RET

001B                    ECHO     ENDP

001B                    LIB      ENDS

                                 END
```

## Using the Video BIOS Function Calls

In addition to the DOS function call INT 21H, we also have video BIOS (*basic I/O system*) function calls at INT 10H. The DOS function calls allow a key to be read and a character to be displayed with ease, but the cursor is difficult to position at the desired screen location. The video BIOS function calls allow more complete control over the video display than do the DOS function calls. The video BIOS function calls are also more efficient and require less time to execute than the DOS function calls.

*Cursor position.* Before any information is placed on the video screen, the position of the cursor should be known. This allows the screen to be cleared and the displayed information to be placed at any location on the video screen. The BIOS INT 10H function number 03H allows the cursor position to be read from the video interface. The BIOS INT 10H function number 02H allows the cursor to be placed at any screen position. Table 7–2 shows the contents of various registers for both video BIOS INT 10H function calls 02H and 03H.

**TABLE 7–2** BIOS function INT 10H

| *AH* | *Description* | *Parameters* |
|---|---|---|
| 02H | Sets cursor position | DH = Row<br>DL = Column<br>BH = Page number |
| 03H | Reads cursor position | BH = Page number<br>DH = Row<br>DL = Column |

The page number in register BH should be 0 before setting the cursor position. Most software does not access the other pages (1–7) for the video display. If the display adapter is a VGA display, always use page 0. The page number is often ignored after a cursor read. The 0 page is available in the CGA (*color graphics adapter*), EGA (*enhanced graphics adapter*), and VGA (*variable graphics array*) text modes of operation. The other pages are available in some VGA and EGA modes and all CGA modes of operation.

The cursor position assumes that the left-hand page column is column 0 progressing across a line to column 79. The row number corresponds to the character line number on the screen. Row 0 is the uppermost line while row 24 is the last line on the screen. This assumes that the text mode selected for the video adapter is 80 characters per line by 25 lines (80 × 25). Other text modes are available, such as 40 × 25 and 96 × 43.

**EXAMPLE 7–16**

```
0000                    CODE    SEGMENT 'CODE'
                                ASSUME  CS:CODE

0000                    MAIN    PROC FAR

                        HOME    MACRO
                                MOV  AH,2               ;;set cursor position
                                MOV  AL,0               ;;page 0
                                MOV  DX,0               ;;row 0, column 0
                                INT  10H
                                ENDM

                                HOME                    ;use HOME macro
0000 B4 02              1       MOV  AH,2
0002 B7 00              1       MOV  BH,0
0004 BA 0000            1       MOV  DX,0
0007 CD 10              1       INT  10H

0009 B9 07D0                    MOV  CX,2000
000C B4 06                      MOV  AH,6
000E B2 20                      MOV  DL,''              ;space
0010                    MAIN1:
0010 CD 21                      INT  21H                ;display space
0012 E2 FC                      LOOP MAIN1              ;repeat 1920 times

                                HOME                    ;use HOME macro
```

```
0014 B4 02        1             MOV   AH,2
0016 B7 00        1             MOV   BH,0
0018 BA 0000      1             MOV   DX,0
001B CD 10        1             INT   10H

001D B4 4C                      MOV   AH,4CH          ;exit to DOS
001F CD 21                      INT   21H

0021                    MAIN    ENDP
0021                    CODE    ENDS

                                END   MAIN
```

Example 7–16 shows how the INT 10H BIOS function call is used to clear the video screen. This is just one method of clearing the screen. Notice that the first function call positions the cursor to row 0 and column 0, which is called the *home* position. Next we use the DOS function call to write 2000 (80 characters per line × 25 character lines) blank spaces (20H) on the video display, then we again home the cursor.

If this example is assembled, linked, and executed, a major problem surfaces. This program is far too slow to be useful in most cases, because the DOS INT 21H call requires too much time to execute. To correct this situation, another video BIOS function call is used. We can use the scroll function, INT 10H function (06H), to clear the screen at a much higher speed.

**EXAMPLE 7–17**

```
0000                    CODE    SEGMENT 'CODE'
                                ASSUME  CS:CODE

0000                    MAIN    PROC FAR

                        HOME    MACRO
                                MOV   AH,2            ;;set cursor position
                                MOV   BH,0            ;;page 0
                                MOV   DX,0            ;;row 0, column 0
                                INT   10H
                                ENDM

0000 32 FF                      XOR   BH,BH           ;page 0
0002 B4 08                      MOV   AH,8            ;read attributes
0004 CD 10                      INT   10H

0006 8A DF                      MOV   BL,BH
0008 8A FC                      MOV   BH,AH
000A 2B C9                      SUB   CX,CX
000C BA 194F                    MOV   DX,194FH
000F B8 0600                    MOV   AX,0600H        ;clear page 0
0012 CD 10                      INT   10H

                                HOME
0014 B4 02        1             MOV   AH,2
0016 B7 00        1             MOV   BH,0
0018 BA 0000      1             MOV   DX,0
001B CD 10        1             INT   10H

001D B4 4C                      MOV   AH,4CH          ;exit to DOS
001F CD 21                      INT   21H
```

```
0021                    MAIN    ENDP
0021                    CODE    ENDS
                                END  MAIN
```

Function 06H is used with a 00H in AL to blank the entire screen. This allows Example 7–16 to be rewritten so the screen clears at a much higher speed. See Example 7–17 for an improved version of the clear and home cursor program. Here the video BIOS function call number 08H reads the character attributes for blanking the screen. Next, they are positioned in the correct registers and DX is loaded with the screen size, 4FH (79) and 19H (25). If this program is assembled, linked, executed, and compared with Example 7–16, there is a vast improvement in the speed at which the screen is cleared. Please refer to Appendix A for other video BIOS INT 10H function calls that may prove useful in your applications. Also listed in Appendix A are all the INT functions available in most computers.

## Direct Memory Access for the Video Text Screen

The least preferred, but quickest, method of accessing the video text display is a direct access to the video text memory. The video memory for text modes 0–3 (the most commonly used modes) begins at memory location B800:0000. Each video display text character is stored in two memory locations. The first location contains the ASCII-coded character and the second contains the video display attribute of the character. The problem with direct access is that the cursor does not move, but it is useful if the screen data must be saved and reloaded without affecting the cursor.

The attribute byte is illustrated in Figure 7–1. The various colors used in the attribute byte are listed in Table 7–3. Note that the text can have 16 colors and the background can only have eight. To illustrate the way that direct access to video text memory functions refer to Example 7–18. This example sets the video mode to 3 using an INT 10H mode set function. It then proceeds to blank the display by directly accessing the video memory. After the screen is blanked, it displays a message across the first line of the screen using a bright green, blinking text on a red background to illustrate the use of the background color and blink options found in the attribute byte.

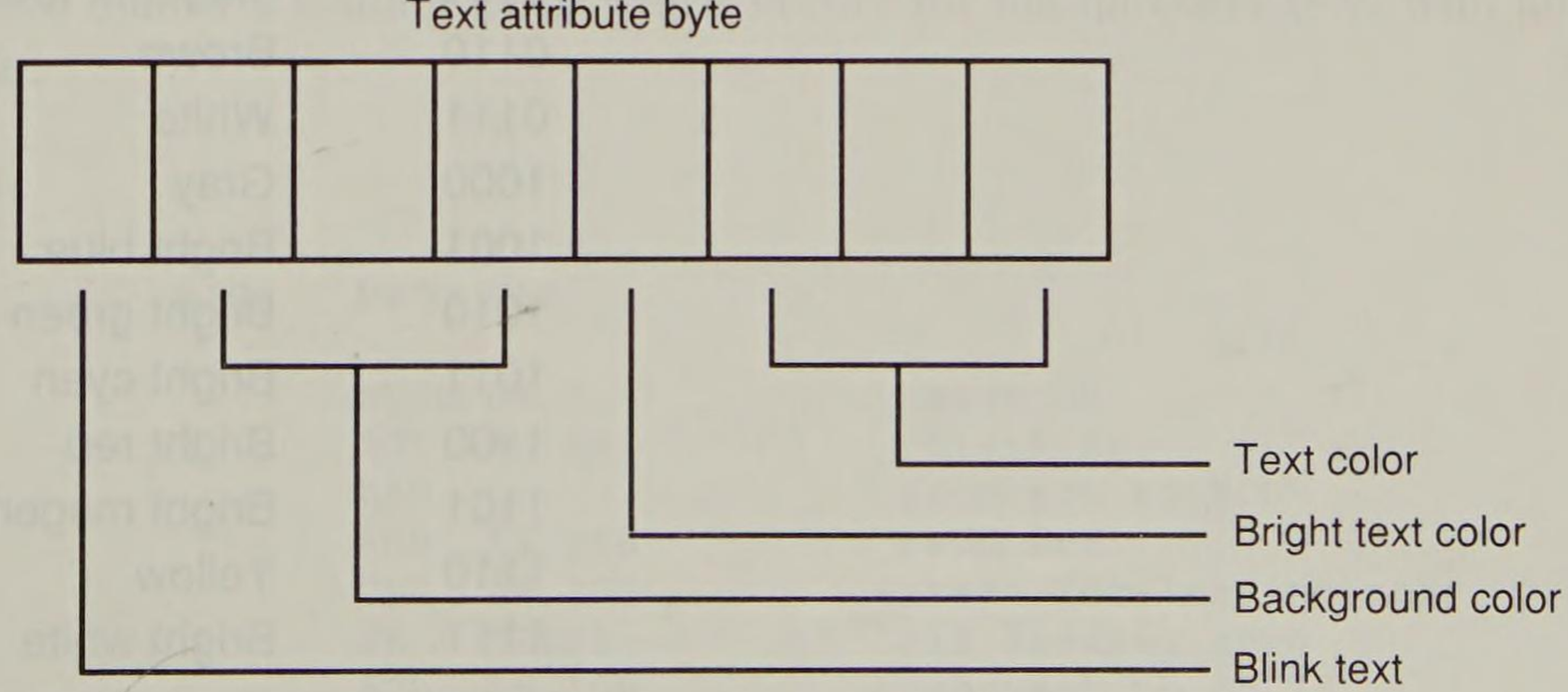



**FIGURE 7–1** The text attribute byte.

**EXAMPLE 7–18**

```
0000                    DATA    SEGMENT

0000 54 68 69 73 20     MES1    DB   'This is a test message.',0
     69 73 20 61 20
     74 65 73 74 20
     6D 65 73 73 61
     67 65 2E 00

0018                    DATA    ENDS

0000                    CODE    SEGMENT 'CODE'
                                ASSUME  CS:CODE,DS:DATA

0000                    MAIN    PROC FAR
0000 B8 ---- R                  MOV  AX,DATA          ;address data segment
0003 8E D8                      MOV  DS,AX
0005 FC                         CLD                   ;select autoincrement

0006 B8 0003                    MOV  AX,0003H         ;select video mode 3
0009 CD 10                      INT  10H              ;this blanks the screen

000B B4 CA                      MOV  AH,11001010B     ;green text, red background
000D BE 0000 R                  MOV  SI,OFFSET MES1

0010 BB B800                    MOV  BX,0B800H        ;address video memory
0013 8E C3                      MOV  ES,BX
0015 33 FF                      XOR  DI,DI
0017                    MAIN1:                        ;display MES1
0017 AC                         LODSB                 ;get ASCII
0018 0A C0                      OR   AL,AL
001A 74 03                      JZ   MAIN2            ;if finished
```

**TABLE 7–3** Colors used in the attribute byte

| *Code* | *Color* |
|---|---|
| 0000 | Black |
| 0001 | Blue |
| 0010 | Green |
| 0011 | Cyan (blue-green) |
| 0100 | Red |
| 0101 | Magenta (purple) |
| 0110 | Brown |
| 0111 | White |
| 1000 | Gray |
| 1001 | Bright blue |
| 1010 | Bright green |
| 1011 | Bright cyan |
| 1100 | Bright red |
| 1101 | Bright magenta |
| 1110 | Yellow |
| 1111 | Bright white |

```
001C AB                     STOSW               ;store ASCII and attribute
001D EB F8                  JMP  MAIN1
001F                  MAIN2:
001F B4 06                  MOV  AH,6           ;wait for any key
0021 B2 FF                  MOV  DL,0FFH
0023 CD 21                  INT  21H
0025 74 F8                  JZ   MAIN2

0027 B8 4C00                MOV  AX,4C00H       ;exit to DOS
002A CD 21                  INT  21H

002C                  MAIN  ENDP
002C                  CODE  ENDS

                            END  MAIN
```

## 7–3 DATA CONVERSIONS

In computer systems, data is seldom in the correct form. One main task of the system is to convert data from one form to another. This section of the chapter describes conversions between binary and ASCII. Binary data are removed from a register or memory and converted to ASCII for the video display. In many cases, ASCII data are converted to binary as they are typed on the keyboard. We also explain converting between ASCII and hexadecimal data.

### Converting from Binary to ASCII

Conversion from binary to ASCII is accomplished in two ways: (1) by the AAM instruction if the number is less than 100, and (2) by a series of decimal divisions (divide by 10). Both techniques are presented in this section.

The AAM instruction converts the value in AX into a two-digit unpacked BCD number in AX. If the number in AX is 0062H (98 decimal) before AAM executes, AX contains a 0908H after AAM executes. This is not ASCII code, but it is converted to ASCII code by adding a 3030H to AX. Example 7–19 illustrates a procedure that processes the binary value in AL (0–99) and displays it on the video screen as a decimal number. This procedure blanks a leading zero, which occurs for the numbers 0–9, with an ASCII space code.

**EXAMPLE 7–19**

```
0000                  DISP  PROC FAR

0000 52                     PUSH DX             ;save DX
0001 32 E4                  XOR  AH,AH          ;blank AH
0003 D4 0A                  AAM                 ;convert to BCD
0005 80 C4 20               ADD  AH,20H         ;add 20H
0008 80 FC 20               CMP  AH,20H         ;test for leading zero
000B 74 03                  JE   DISP1          ;if leading zero
000D 80 C4 10               ADD  AH,10H         ;convert to ASCII
```

```
0010                    DISP1:
0010 50                         PUSH AX
0011 8A D4                      MOV  DL,AH          ;display first digit
0013 B4 06                      MOV  AH,6
0015 CD 21                      INT  21H
0017 58                         POP  AX
0018 04 30                      ADD  AL,30H         ;convert to ASCII
001A 8A D0                      MOV  DL,AL
001C B4 06                      MOV  AH,6           ;display second digit
001E CD 21                      INT  21H
0020 5A                         POP  DX             ;restore DX
0021 CB                         RET
0022                    DISP    ENDP
```

The reason that AAM converts any number between 0 and 99 to a two-digit unpacked BCD number is because it divides AX by 10. The result is left in AX, with AH containing the quotient and AL the remainder. This same scheme of dividing by ten can be expanded to convert any whole number from binary to an ASCII-coded character string that can be displayed on the video screen. The algorithm for converting from binary to ASCII is:

1. Divide by 10 and save the remainder on the stack as a significant BCD digit.
2. Repeat Step 1 until the quotient is a 0.
3. Retrieve each remainder and add a 30H to convert to ASCII before displaying or printing.

Example 7–20 shows how the unsigned 16-bit contents of AX are converted to ASCII and displayed on the video screen as an unsigned integer. Here, we divide AX by 10 and save the remainder on the stack after each division for later conversion to ASCII. The reason that data are stored on the stack is because the least-significant digit is returned first by the division. The stack is used to reverse the order of the data so it can be displayed correctly from the most-significant to the least-significant digit. After all the digits have been converted by division, the result is displayed on the video screen by removing the remainders from the stack and converting them to ASCII code. This procedure also blanks any leading zeros that occur.

**EXAMPLE 7–20**

```
0000                    DISPX  PROC FAR

0000 52                        PUSH DX              ;save BX, CX, and DX
0001 51                        PUSH CX
0002 53                        PUSH BX

0003 33 C9                     XOR  CX,CX           ;clear CX
0005 BB 000A                   MOV  BX,10           ;load 10
0008                    DISPX1:
0008 33 D2                     XOR  DX,DX           ;clear DX
000A F7 F3                     DIV  BX
000C 52                        PUSH DX              ;save remainder
000D 41                        INC  CX              ;count remainder
000E 0B C0                     OR   AX,AX           ;test quotient
0010 75 F6                     JNZ  DISPX1          ;if not zero
0012                    DISPX2:
0012 5A                        POP  DX              ;display number
```

```
0013 B4 06                      MOV   AH,6
0015 80 C2 30                   ADD   DL,30H          ;convert to ASCII
0018 CD 21                      INT   21H
001A E2 F6                      LOOP  DISPX2          ;repeat

001C 5B                         POP   BX              ;restore BX, CX, and DX
001D 59                         POP   CX
001E 5A                         POP   DX
001F CB                         RET

0020                    DISPX   ENDP
```

## Converting from ASCII to Binary

Conversions from ASCII to binary usually start with keyboard data entry. If a single key is typed, the conversion is accomplished by subtracting a 30H from the number. If more than one key is typed, conversion from ASCII to binary still requires that 30H is subtracted, but there is one additional step. After subtracting 30H, the number is added to the result after the prior result is first multiplied by a 10. The algorithm used to convert ASCII to binary is:

1. Begin with a binary result of 0.
2. Subtract 30H from the character typed on the keyboard to convert it to BCD.
3. Multiply the binary result by 10 and then add the new BCD digit to the result.
4. Repeat Steps 2 and 3 until the character typed is not an ASCII-coded number of 30H–39H.

**EXAMPLE 7–21**

```
0000                    READN   PROC FAR

0000 53                         PUSH BX               ;save BX and CX
0001 51                         PUSH CX
0002 B9 000A                    MOV  CX,10            ;load 10
0005 33 DB                      XOR  BX,BX            ;clear result
0007                    READN1:
0007 B4 06                      MOV  AH,6             ;read key
0009 B2 FF                      MOV  DL,0FFH
000B CD 21                      INT  21H
000D 74 F8                      JE   READN1           ;wait for key

000F 3C 30                      CMP  AL,'0'           ;test against 0
0011 72 18                      JB   READN2           ;if below 0
0013 3C 39                      CMP  AL,'9'           ;test against 9
0015 77 14                      JA   READN2           ;if above 9

0017 8A D0                      MOV  DL,AL            ;echo key
0019 CD 21                      INT  21H

001B 2C 30                      SUB  AL,'0'           ;convert to BCD

001D 50                         PUSH AX
001E 8B C3                      MOV  AX,BX            ;multiply by 10
0020 F7 E1                      MUL  CX
0022 8B D8                      MOV  BX,AX            ;save product
0024 58                         POP  AX
0025 32 E4                      XOR  AH,AH
0027 03 D8                      ADD  BX,AX            ;add BCD to product
```

```
0029 EB DC                      JMP   READN1          ;repeat
002B                        READN2:
002B 8B C3                      MOV   AX,BX           ;move binary to AX
002D 59                         POP   CX              ;restore BX and CX
002E 5B                         POP   BX
002F CB                         RET

0030                        READN   ENDP
```

Example 7–21 illustrates a procedure that implements the ASCII-to-binary conversion algorithm. Here the binary number returns in the AX register as a 16-bit result. If a larger result is required, the procedure must be reworked for 32-bit arithmetic. Each time this procedure is called, it reads a number from the keyboard until any key other than 0 through 9 is typed. It then returns with the binary equivalent in the AX register.

Signed numbers can also be converted if we first test for a sign and then use Example 7–21 to convert the number to binary. After the conversion, if the sign was minus, the result is two's complemented with NEG to create a negative version of the binary number.

## Displaying and Reading Hexadecimal Data

Hexadecimal data is easier to read from the keyboard and display than decimal data. Hexadecimal data is not used at the applications level, but at the system level. System level data is often hexadecimal and must be either displayed in hexadecimal form or read from the keyboard as hexadecimal data.

*Reading Hexadecimal Data.* Hexadecimal data appear as 0 to 9 and A to F. The ASCII codes obtained from the keyboard for hexadecimal data are 30H to 39H for the numbers 0 through 9, and 41H to 46H (A–F) or 61H to 66H (a–f) for the letters. To be useful, a procedure that reads hexadecimal data must be able to accept both lowercase (a–f) and uppercase (A–F) letters.

**EXAMPLE 7–22**

```
0000                        CONV    PROC NEAR

0000 3C 39                      CMP   AL,'9'
0002 76 08                      JBE   CONV2           ;if a number
0004 3C 61                      CMP   AL,'a'
0006 72 02                      JB    CONV1           ;if uppercase
0008 2C 20                      SUB   AL,20H          ;convert to uppercase
000A                        CONV1:
000A 2C 07                      SUB   AL,7
000C                        CONV2:
000C 2C 30                      SUB   AL,'0'
000E C3                         RET

000F                        CONV    ENDP

000F                        READH   PROC FAR

000F 51                         PUSH CX               ;save BX and CX
0010 53                         PUSH BX
0011 B9 0004                    MOV   CX,4            ;load shift count
0014 8B F1                      MOV   SI,CX           ;load count
```

```
0016 33 DB                  XOR  BX,BX          ;clear result
0018                READH1:
0018 B4 06                  MOV  AH,6           ;read key
001A B2 FF                  MOV  DL,0FFH
001C CD 21                  INT  21H
001E 74 F8                  JE   READH1         ;wait for key
0020 8A D0                  MOV  DL,AL          ;echo
0022 CD 21                  INT  21H
0024 E8 0000 R              CALL CONV           ;convert to hexadecimal
0027 D3 E3                  SHL  BX,CL          ;shift result
0029 02 D8                  ADD  BL,AL          ;add AL to result
002B 4E                     DEC  SI
002C 75 EA                  JNZ  READH1         ;repeat 4 times
002E 5B                     POP  BX             ;restore BX and CX
002F 59                     POP  CX
0030 CB                     RET

0031                READH   ENDP
```

Example 7–22 shows two procedures: one converts the contents of the data in AL from an ASCII-coded character to a single hexadecimal digit, the other reads a 4-digit hexadecimal number from the keyboard and returns with it in register AX. The second procedure can be modified to read any size hexadecimal number from the keyboard. Notice how the letters a through f are converted to uppercase (subtract 20H) in the first procedure.

*Displaying Hexadecimal Data.* To display hexadecimal data, a number is separated into 4-bit segments that are converted into hexadecimal digits. Conversion is accomplished by adding a 30H to the numbers 0 to 9, and a 37H to the letters A to F.

**EXAMPLE 7–23**

```
0000                DISPH   PROC FAR

0000 51                     PUSH CX             ;save CX and DX
0001 52                     PUSH DX
0002 B1 04                  MOV  CL,4           ;load rotate count
0004 B5 04                  MOV  CH,4           ;load digit count
0006                DISPH1:
0006 D3 C0                  ROL  AX,CL          ;position number
0008 50                     PUSH AX             ;save it
0009 24 0F                  AND  AL,0FH         ;get hex digit
000B 04 30                  ADD  AL,30H         ;adjust it
000D 3C 39                  CMP  AL,'9'         ;test against 9
000F 76 02                  JBE  DISPH2         ;if 0 - 9
0011 04 07                  ADD  AL,7           ;adjust it
0013                DISPH2:
0013 B4 06                  MOV  AH,6
0015 8A D0                  MOV  DL,AL
0017 CD 21                  INT  21H            ;display digit
0019 58                     POP  AX             ;restore AX
001A FE CD                  DEC  CH
001C 75 E8                  JNZ  DISPH1         ;repeat

001E 5A                     POP  DX             ;restore CX and DX
001F 59                     POP  CX
0020 CB                     RET

0021                DISPH   ENDP
```

A procedure that displays the contents of the AX register on the video display appears in Example 7-23. Here the number is rotated left so the left-most digit is displayed first. Because AX contains a 4-digit hexadecimal number, the procedure displays four hexadecimal digits. This procedure can be modified to display other-width hexadecimal numbers.

## Using Lookup Tables for Data Conversions

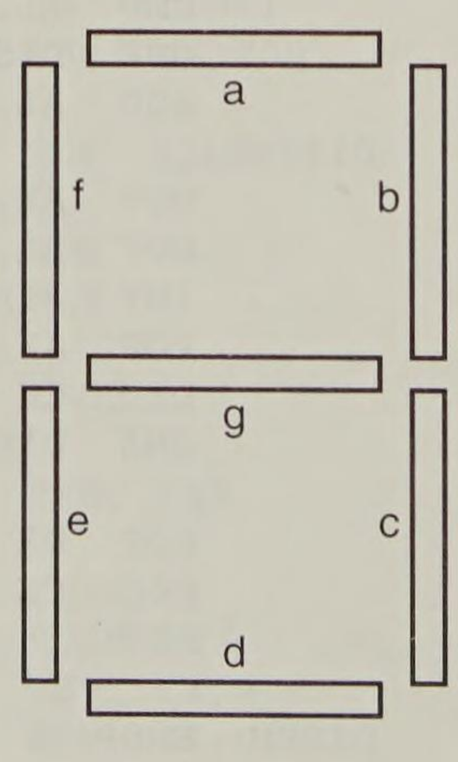
Lookup tables are often used to convert from one data form to another. A lookup table is placed in the memory as a list of data that is referenced by a procedure to perform conversions. In many lookup tables the XLAT instruction is used to look up data. This requires that the table contains 8-bit wide data and its length is less than or equal to 256 bytes.

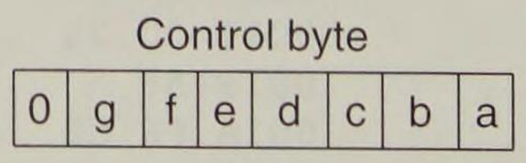
*Converting from BCD to 7-Segment Code.* One simple application that uses a lookup table is BCD to 7-segment code conversion. Example 7–24 illustrates a lookup table that contains the 7-segment codes for the numbers 0 through 9. These codes are used with the 7-segment display pictured in Figure 7–2. This 7-segment display uses active high (logic 1) inputs to light a segment. The code is arranged so that the *a* segment is in bit position 0 and the *g* segment is in bit position 6. Bit position seven is zero in this example, but can be used for displaying a decimal point.

**EXAMPLE 7–24**

```
0000                    SEG7    PROC FAR

0000 53                         PUSH BX
0001 BB 0008 R                  MOV  BX,OFFSET TABLE
0004 2E: D7                     XLAT  CS:TABLE          ;see text
0006 5B                         POP  BX
0007 CB                         RET

0008 3F                 TABLE   DB   3FH               ;0
0009 06                         DB   6                 ;1
000A 5B                         DB   5BH               ;2
000B 4F                         DB   4FH               ;3
000C 66                         DB   66H               ;4
000D 6D                         DB   6DH               ;5
000E 7D                         DB   7DH               ;6
```

**FIGURE 7–2** The 7-segment display.



```
000F 07                         DB    7                 ;7
0010 7F                         DB    7FH               ;8
0011 6F                         DB    6FH               ;9

0012                    SEG7    ENDP
```

The procedure that performs the conversion contains only two instructions and assumes that AL contains the BCD digit to be converted to 7-segment code. One of the instructions addresses the lookup table by loading its address into BX, and the other instruction performs the conversion and returns the 7-segment code in AL.

Because the lookup table is located in the code segment, and the XLAT instruction accesses the data segment by default, the XLAT instruction includes a segment override. Notice that a dummy operand (TABLE) is added to the XLAT instruction so the (CS:) code segment override prefix can be added to the instruction. Normally XLAT does not contain an operand unless its default segment must be overridden. The LODS and MOVS instructions can also be overridden in the same manner as XLAT by using a dummy operand.

*Using a Lookup Table to Access ASCII Data.* Some programming techniques require that numeric codes are converted to ASCII character strings. For example, suppose you need to display the days of the week for a calendar program. Because the number of ASCII characters in each day is different, some type of lookup table must be used to reference the ASCII-coded days of the week.

**EXAMPLE 7–25**

```
0000                    DAYS    PROC FAR

0000 52                         PUSH DX                ;save DX and SI
0001 56                         PUSH SI
0002 BE 001B R                  MOV  SI,OFFSET DTAB    ;address DTAB
0005 32 E4                      XOR  AH,AH             ;clear AH
0007 03 C0                      ADD  AX,AX             ;double AX
0009 03 F0                      ADD  SI,AX             ;modify table address
000B 2E: 8B 14                  MOV  DX,CS:[SI]        ;get string address
000E 8C C8                      MOV  AX,CS             ;change data segment
0010 1E                         PUSH DS
0011 8E D8                      MOV  DS,AX
0013 B4 09                      MOV  AH,9
0015 CD 21                      INT  21H               ;display string
0017 1F                         POP  DS
0018 5E                         POP  SI                ;restore DX and SI
0019 5A                         POP  DX
001A CB                         RET

001B 0029 R 0031 R      DTAB    DW   SUN,MON,TUE,WED,THU,FRI,SAT
     0039 R 0042 R
     004D R 0057 R
     005F R

0029 53 75 6E 64 61     SUN     DB   'Sunday $'
     79 20 24

0031 4D 6F 6E 64 61     MON     DB   'Monday $'
     79 20 24
```

```
0039 54 75 65 73 64  TUE    DB   'Tuesday $'
     61 79 20 24

0042 57 65 64 6E 65  WED    DB   'Wednesday $'
     73 64 61 79 20
     24

004D 54 68 75 72 73  THU    DB   'Thursday $'
     64 61 79 20 24

0057 46 72 69 64 61  FRI    DB   'Friday $'
     79 20 24

005F 53 61 74 75 72  SAT    DB   'Saturday $'
     64 61 79 20 24

0069                 DAYS   ENDP
```

Example 7–25 shows a table that references ASCII-coded character strings located in the code segment. Each character string contains an ASCII-coded day of the week. The table references each day of the week. The procedure that accesses the day of the week uses the AL register and the numbers 0 to 6 to refer to Sunday through Saturday. If AL contains a 2 when this procedure is called, the word *"Tuesday"* is displayed on the video screen.

This procedure first accesses the table by loading the table address into the SI register. Next, the number in AL is converted into a 16-bit number and doubled, because each table entry is two bytes in length. This index is then added to SI to address the correct entry in the lookup table. The address of the ASCII character string is now loaded from the lookup table into DX by the MOV DX,CS:[SI] instruction.

Before the INT 21H DOS function is called, the DS register is placed on the stack and loaded with the segment address of CS. This allows DOS function number 09H (display a string) to be used to display the day of the week. Recall that function 09H requires the address of the character string to be loaded into DS:DX before the function call. This procedure converts the numbers 0 to 6 to the seven days of the week.

## An Example Program Using Data Conversions

A program example is required to combine some of the data conversion DOS functions discussed thus far. Suppose you must display the time and date on the video screen. An example program (see Example 7–26) displays the time as 10:45 P.M. and the date as Tuesday, May 14, 1993. The program is short because it calls a procedure that displays the time and a second procedure that displays the date.

**EXAMPLE 7–26** (pp. 230-234)

```
0000                 STAC   SEGMENT  STACK

0000 0100[                  DW   256 DUP (?)      ;set up stack
          ????
               ]

0200                 STAC   ENDS

0000                 DAT    SEGMENT
```

```
0000 0026 R 002F R   DAY     DW    SUN,MON,TUE,WED,THU,FRI,SAT
       0038 R 0042 R
       004E R 0059 R
       0062 R
000E 006D R 0076 R   MONT    DW    JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT
       0080 R 0087 R
       008E R 0093 R
       0099 R 009F R
       00A7 R 00B2 R
       00BB R 00C5 R         DW    NOV,DCE

0026 53 75 6E 64 61  SUN     DB    'Sunday, $'
       79 2C 20 24
002F 4D 6F 6E 64 61  MON     DB    'Monday, $'
       79 2C 20 24
0038 54 75 65 73 64  TUE     DB    'Tuesday, $'
       79 79 2C 20 24
0042 57 65 64 6E 65  WED     DB    'Wednesday, $'
       73 64 61 79 2C
       20 24
004E 54 68 75 72 73  THU     DB    'Thursday, $'
       64 61 79 2C 20
       24
0059 46 72 69 64 61  FRI     DB    'Friday, $'
       79 2C 20 24
0062 53 61 74 75 72  SAT     DB    'Saturday, $'
       64 61 79 2C 20
       24
006D 4A 61 6E 75 61  JAN     DB    'January $'
       72 79 20 24
0076 46 65 62 72 75  FEB     DB    'February $'
       61 72 79 20 24
0080 4D 61 72 63 68  MAR     DB    'March $'
       20 24
0087 41 70 72 69 6C  APR     DB    'April $'
       20 24
008E 4D 61 79 20 24  MAY     DB    'May $'
0093 4A 75 6E 65 20  JUN     DB    'June $'
       24
0099 4A 75 6C 79 20  JUL     DB    'July $'
       24
009F 41 75 67 75 73  AUG     DB    'August $'
       74 20 24
00A7 53 65 70 74 65  SEP     DB    'September $'
       6D 62 65 72 20
       24
00B2 4F 63 74 6F 62  OCT     DB    'October $'
       65 72 20 24
00BB 4E 6F 76 65 6D  NOV     DB    'November $'
       62 65 72 20 24
00C5 44 65 63 65 6D  DCE     DB    'December $'
       62 65 72 20 24

00CF 0D 0A 24        CRLF    DB    13,10,'$'
00D2 2E 4D 2E 20 20  MES1    DB    '.M.  $'
       24
00D8 2C 20 31 39 24  MES2    DB    ', 19$'
00DD 2C 20 32 30 24  MES3    DB    ', 20$'

00E2                 DAT     ENDS

0000                 COD     SEGMENT 'CODE'
```

```
                                ASSUME CS:COD,DS:DAT,SS:STAC

0000                    MAIN    PROC FAR                ;main program

0000 B8 ---- R                  MOV  AX,DAT             ;load DS
0003 8E D8                      MOV  DS,AX

0005 BA 00CF R                  MOV  DX,OFFSET CRLF     ;get new line
0008 B4 09                      MOV  AH,9
000A CD 21                      INT  21H

000C E8 001D R                  CALL TIMES              ;display time
000F E8 007E R                  CALL DATES              ;display date

0012 BA 00CF R                  MOV  DX,OFFSET CRLF     ;get new line
0015 B4 09                      MOV  AH,9
0017 CD 21                      INT  21H

0019 B4 4C                      MOV  AH,4CH             ;exit to DOS
001B CD 21                      INT  21H

001D                    MAIN    ENDP
001D                    TIMES   PROC NEAR               ;display time XX:XX A.M.

001D B4 2C                      MOV  AH,2CH             ;get time
001F CD 21                      INT  21H

0021 B7 41                      MOV  BH,'A'             ;set AM

0023 80 FD 0C                   CMP  CH,12              ;test against 12
0026 72 05                      JB   TIMES1             ;if AM

0028 B7 50                      MOV  BH,'P'             ;set PM
002A 80 ED 0C                   SUB  CH,12              ;adjust time
002D                    TIMES1:
002D 0A ED                      OR   CH,CH              ;test for 0 hours
002F 75 02                      JNE  TIMES2             ;if not 0 hours
0031 B5 0C                      MOV  CH,12              ;replace with 12 hours
0033                    TIMES2:
0033 8A C5                      MOV  AL,CH              ;get hours
0035 32 E4                      XOR  AH,AH              ;clear AH
0037 D4 0A                      AAM                     ;convert to BCD
0039 0A E4                      OR   AH,AH              ;test tens of hours
003B 74 09                      JZ   TIMES3             ;if no tens of hours
003D 50                         PUSH AX
003E 8A C4                      MOV  AL,AH
0040 04 30                      ADD  AL,'0'             ;convert to ASCII
0042 E8 0075 R                  CALL DISP               ;display tens of hours
0045 58                         POP  AX
0046                    TIMES3:
0046 04 30                      ADD  AL,'0'             ;convert to ASCII
0048 E8 0075 R                  CALL DISP               ;display units of hours
004B B0 3A                      MOV  AL,':'             ;display colon
004D E8 0075 R                  CALL DISP

0050 8A C1                      MOV  AL,CL              ;get minutes
0052 32 E4                      XOR  AH,AH              ;clear AH
0054 D4 0A                      AAM                     ;convert to BCD

0056 05 3030                    ADD  AX,3030H           ;convert to ASCII
0059 50                         PUSH AX
005A 8A C4                      MOV  AL,AH
```

```
005C E8 0075 R              CALL DISP           ;display tens of minutes
005F 58                     POP  AX
0060 E8 0075 R              CALL DISP           ;display units of minutes

0063 B0 20                  MOV  AL,' '         ;display space
0065 E8 0075 R              CALL DISP

0068 8A C7                  MOV  AL,BH          ;display A or P
006A E8 0075 R              CALL DISP

006D BA 00D2 R              MOV  DX,OFFSET MES1 ;display .M.
0070 B4 09                  MOV  AH,9
0072 CD 21                  INT  21H
0074 C3                     RET

0075                TIMES   ENDP

0075                DISP    PROC NEAR           ;display ASCII

0075 50                     PUSH AX
0076 B4 06                  MOV  AH,6
0078 8A D0                  MOV  DL,AL
007A CD 21                  INT  21H
007C 58                     POP  AX
007D C3                     RET

007E                DISP    ENDP

007E                DATES   PROC NEAR           ;display date

007E B4 2A                  MOV  AH,2AH         ;get date
0080 CD 21                  INT  21H
0082 52                     PUSH DX             ;save month and day

0083 32 E4                  XOR  AH,AH          ;clear AH
0085 03 C0                  ADD  AX,AX          ;double AX
0087 BE 0000 R              MOV  SI,OFFSET DAY  ;address day table
008A 03 F0                  ADD  SI,AX
008C 8B 14                  MOV  DX,[SI]        ;get string address
008E B4 09                  MOV  AH,9
0090 CD 21                  INT  21H            ;display day

0092 5A                     POP  DX
0093 52                     PUSH DX
0094 8A C6                  MOV  AL,DH          ;get month
0096 FE C8                  DEC  AL
0098 32 E4                  XOR  AH,AH
009A 03 C0                  ADD  AX,AX
009C BE 000E R              MOV  SI,OFFSET MONT ;address month table
009F 03 F0                  ADD  SI,AX
00A1 8B 14                  MOV  DX,[SI]        ;get string address
00A3 B4 09                  MOV  AH,9
00A5 CD 21                  INT  21H            ;display month

00A7 5A                     POP  DX             ;get day
00A8 8A C2                  MOV  AL,DL
00AA 32 E4                  XOR  AH,AH          ;clear AH
00AC D4 0A                  AAM                 ;convert to BCD
00AE 0A E4                  OR   AH,AH          ;test tens of day
00B0 74 09                  JZ   DATES1         ;if zero
00B2 50                     PUSH AX
00B3 8A C4                  MOV  AL,AH
```

```
00B5 04 30                      ADD  AL,'0'          ;convert to ASCII
00B7 E8 0075 R                  CALL DISP            ;display tens of day
00BA 58                         POP  AX
00BB                      DATES1:
00BB 04 30                      ADD  AL,'0'          ;convert to ASCII
00BD E8 0075 R                  CALL DISP            ;display units of day
00C0 BA 00D8 R                  MOV  DX,OFFSET MES2
00C3 81 F9 07D0                 CMP  CX,2000         ;test for year 2000
00C7 72 06                      JB   DATES2          ;if year 19XX
00C9 83 E9 64                   SUB  CX,100
00CC BA 00DD R                  MOV  DX,OFFSET MES3
00CF                      DATES2:
00CF B4 09                      MOV  AH,9            ;display 19 or 20
00D1 CD 21                      INT  21H
00D3 81 E9 076C                 SUB  CX,1900         ;adjust year
00D7 8B C1                      MOV  AX,CX
00D9 D4 0A                      AAM                  ;convert to BCD
00DB 05 3030                    ADD  AX,3030H        ;convert to ASCII
00DE 50                         PUSH AX
00DF 8A C4                      MOV  AL,AH
00E1 E8 0075 R                  CALL DISP            ;display tens of year
00E4 58                         POP  AX
00E5 E8 0075 R                  CALL DISP            ;display units of year
00E8 C3                         RET

00E9                      DATES ENDP

00E9                      COD   ENDS

                                END  MAIN
```

The time is available from DOS by using an INT 21H function call number 2CH. This function returns with the hours in CH and minutes in CL. Also available are seconds in DH and hundredths of seconds in DL. The date is available by using INT 21H function call number 2AH. This leaves the day of the week in AL, the year in CX in binary, the day of the month in DH, and the month in DL.

The example procedure uses two ASCII lookup tables that convert the day and month to ASCII character strings. It also uses the AAM instruction to convert from binary to BCD for the time and date. Date display is handled in two ways: by character string (function 09H) and by single character (function 06H).

The memory consists of three segments: stack (STAC), data (DAT) and code (COD). The data segment contains the characters strings used with the procedures that display time and date. The code segment contains MAIN, TIMES, DATES, and DISP procedures. The MAIN procedure is a FAR procedure because it is the program or main module. The other procedures are NEAR or local procedures used by MAIN.

## 7–4 VGA GRAPHIC DISPLAYS

Because most modern systems contain some form of a VGA graphic display, a section on this topic is included as an introduction to graphic display systems. This topic alone could consume an entire textbook, so only the basics are explained and applied here.

Video memory exists at locations A0000H through AFFFFH for the VGA graphics modes. (The text modes use video memory beginning at location B8000H or B0000H depending upon the mode selected.) The VGA graphics memory is 64K bytes in size and requires bit planes to address the 153,600 bytes of memory required for a 16-color, 640 × 480 display. Video cards contain 256K bytes or more memory that is addressed in sections (pages) through the 64K-byte memory window at locations A0000H–AFFFFH. Each time a new bit plane is selected, the adapter internally addresses a separate 64K-byte section of memory that appears at locations A0000H–AFFFFH.

When the 256-color, 320 × 200 mode is selected, a byte in the video memory selects a single color (1 of 256) for a single PEL. This graphics display mode (13H) uses 64,000 bytes of memory (slightly less than 64K bytes), which are directly addressed at locations A0000H–AEA5FH. Although this display mode has a lower resolution, it displays 256 colors instead of 16. Note that this area of memory is organized differently than for the 16-color, 640 × 480 VGA display mode. The first byte (A000:0000) holds the upper left PEL instead of eight PELs.

## Programming in the 256-Color, 320 × 200 Mode

Software for the *bit-mapped* 256-color, 320 × 200 display mode is fairly easy to write because each byte represents a single PEL on the video display. Before any information is displayed, the display adapter (assuming VGA is present) must be switched to mode 13H, the 256-color, 320 × 200 mode. This is accomplished with video BIOS INT 10H function 00H. To select a new video mode, place 00H into AH and the new mode number in AL. This is followed by the video BIOS function call INT 10H to select the new mode.

Once in the new mode, a program can begin to display graphics information. Example 7–27 shows a short program that switches the display to the 256-color, 320 × 200 VGA mode and then displays a vertical color bar pattern on the screen to show the 256 colors programmed by default into the VGA adapter. Here 2 PELs are used for each color across the entire width of the display.

**EXAMPLE 7–27**

```
0000                    CODE    SEGMENT 'CODE'
                                ASSUME CS:CODE

0000                    COLOR   PROC FAR

0000 B8 A000                    MOV   AX,0A000H       ;address video memory
0003 8E C0                      MOV   ES,AX
0005 BF 0000                    MOV   DI,0
0008 FC                         CLD                   ;select auto-increment

0009 B4 00                      MOV   AH,0
000B B0 13                      MOV   AL,13H          ;select mode 13H
000D CD 10                      INT   10H             ;this also clears the screen
000F BB 00C8                    MOV   BX,200          ;scan line count
0012                    COLOR1:
0012 B9 00A0                    MOV   CX,320/2        ;column count/2
0015 B0 00                      MOV   AL,0            ;get color number
0017                    COLOR2:
0017 AA                         STOSB                 ;store 2 PELs
```

```
0018 AA                 STOSB
0019 FE C0              INC  AL             ;change to new color
001B E2 FA              LOOP COLOR2         ;repeat for one scan line

001D 4B                 DEC  BX             ;repeat 200 times
001E 75 F2              JNZ  COLOR1
0020                COLOR3:
0020 B4 06              MOV  AH,6           ;wait for any key
0022 B2 FF              MOV  DL,0FFH
0024 CD 21              INT  21H
0026 74 F8              JZ   COLOR3

0028 B4 4C              MOV  AH,4CH         ;exit to DOS after key
002A CD 21              INT  21H

                    COLOR   ENDP

002C                CODE    ENDS

                            END  COLOR
```

What if you want to change the default colors? This is accomplished by directly reprogramming a series of palette registers located on the video display card. There are 256 different locations in the palette that each represent a video display color. Each picture element is composed of the three basic video colors of red, green, and blue. These are the primary colors of light. To display a color, we use a combination of these three primary colors. The palette memory contains three 6-bit numbers (one for each primary color) that change the brightness of each primary color. The program in Example 7–27 uses the default palette, which has colors set so the first 16 colors are compatible with other display modes, the second 16 colors are shades of gray, and the remaining settings are various other colors.

Video BIOS INT 10H functions AH = 10H, AL = 10H, and BX = color number (00H–FFH) allow a palette location (color number) to be selected and changed to CH = green, CL = blue, and DH = red. The color values in CH, CL, and DH are 6-bit numbers ranging from 00H–3FH. A value of 00H is off and 3FH is maximum brightness. For example, a very bright cyan (blue-green) is CH = 3FH, CL = 3FH, and DH = 00H. Varying these color amplitudes and various combinations vary the brightness and hue of each color number.

**EXAMPLE 7–28**

```
0000                CODE    SEGMENT 'CODE'
                            ASSUME CS:CODE

0000                COLOR   PROC FAR

0000 B8 A000                MOV  AX,0A000H      ;address video memory
0003 8E C0                  MOV  ES,AX
0005 BF 0000                MOV  DI,0
0008 FC                     CLD                 ;select auto-increment

0009 B4 00                  MOV  AH,0
000B B0 13                  MOV  AL,13H         ;select mode 13H
000D CD 10                  INT  10H            ;this also clears the screen
```

```
000F BD 0040                    MOV   BP,64             ;count
0012 B6 00                      MOV   DH,0              ;set red to zero
0014 BB 0080                    MOV   BX,80H            ;initialize palette number
0017                    COLORA:
0017 B8 1010                    MOV   AX,1010H          ;change palette
001A B9 0000                    MOV   CX,0              ;no green or blue
001D CD 10                      INT   10H
001F 43                         INC   BX                ;next palette
0020 FE C6                      INC   DH                ;next red
0022 4D                         DEC   BP
0023 75 F2                      JNZ   COLORA            ;repeat 64 times

0025 BB 00C8                    MOV   BX,200            ;scan line count
0028                    COLOR1:
0028 B9 0040                    MOV   CX,320/5          ;column count/5
002B B0 80                      MOV   AL,80H            ;get color number
002D                    COLOR2:
002D AA                         STOSB                   ;store 5 PELs
002E AA                         STOSB
002F AA                         STOSB
0030 AA                         STOSB
0031 AA                         STOSB
0032 FE C0                      INC   AL                ;change to new color
0034 E2 F7                      LOOP  COLOR2            ;repeat for one scan line

0036 4B                         DEC   BX                ;repeat 200 times
0037 75 EF                      JNZ   COLOR1
0039                    COLOR3:

0039 B4 06                      MOV   AH,6              ;wait for any key
003B B2 FF                      MOV   DL,0FFH
003D CD 21                      INT   21H
003F 74 F8                      JZ    COLOR3

0041 B4 4C                      MOV   AH,4CH            ;exit to DOS after key
0043 CD 21                      INT   21H

0045                    COLOR   ENDP

0045                    CODE    ENDS

                                END   COLOR
```

To illustrate how the palette is changed, Example 7–28 contains a program that alters color numbers 80H–BFH to every possible brightness level of red. It then displays a vertical color bar pattern, containing 64 bars, to illustrate each intensity of red. This program can be changed to display other colors, by modifying the values loaded into the palette registers.

To utilize this graphics display, procedures are needed to draw shapes and forms on the screen. This text cannot show all variations of these programs, but will show one at this point. Example 7–29 shows a procedure that draws a box on the 256-color, $320 \times 200$ VGA display. The box is any size and can be placed at any location on the screen. Parameters used to transfer size, color, and location are passed through registers to this procedure. To illustrate the operation of the BOX procedure, a program is also included in the example that draws one box on the video display screen.

**EXAMPLE 7–29**

```
0000                    CODE    SEGMENT 'CODE'
                                ASSUME CS:CODE

0000                    MAIN    PROC FAR
0000 FC                         CLD                     ;select auto-increment

0001 B4 00                      MOV  AH,0
0003 B0 13                      MOV  AL,13H             ;select mode 13H
0005 CD 10                      INT  10H                ;this also clears the screen

0007 B0 02                      MOV  AL,2               ;use color 02H
0009 B9 0064                    MOV  CX,100             ;starting column number
000C BE 000A                    MOV  SI,10              ;starting row number
000F BD 004B                    MOV  BP,75              ;size
0012 E8 000D                    CALL BOX                ;display box
0015                    MAIN1:
0015 B4 06                      MOV  AH,6               ;wait for any key
0017 B2 FF                      MOV  DL,0FFH
0019 CD 21                      INT  21H
001B 74 F8                      JZ   MAIN1

001D B8 4C00                    MOV  AX,4C00H           ;exit to DOS
0020 CD 21                      INT  21H

0022                    MAIN    ENDP

0022                    BOX     PROC NEAR

0022 BB A000                    MOV  BX,0A000H          ;address video RAM
0025 8E C3                      MOV  ES,BX
0027 50                         PUSH AX
0028 B8 0140                    MOV  AX,320             ;find starting PEL
002B F7 E6                      MUL  SI
002D 8B F8                      MOV  DI,AX              ;address start of BOX
002F 03 F9                      ADD  DI,CX
0031 58                         POP  AX
0032 57                         PUSH DI                 ;save start
0033 8B CD                      MOV  CX,BP
0035                    BOX1:
0035 F3/ AA                     REP STOSB               ;draw top line
0037 8B CD                      MOV  CX,BP
0039 83 E9 02                   SUB  CX,2               ;adjust CX
003C                    BOX2:
003C 5F                         POP  DI
003D 81 C7 0140                 ADD  DI,320             ;address next row
0041 57                         PUSH DI
0042 AA                         STOSB                   ;draw PEL
0043 03 FD                      ADD  DI,BP
0045 83 EF 02                   SUB  DI,2
0048 AA                         STOSB                   ;draw PEL
0049 E2 F1                      LOOP BOX2
004B 5F                         POP  DI
004C 81 C7 0140                 ADD  DI,320             ;address last row
0050 8B CD                      MOV  CX,BP
0052 F3/ AA                     REP STOSB
0054 C3                         RET

0055                    BOX     ENDP

0055                    CODE    ENDS

                                END  MAIN
```

## Programming in the 16-Color, 640 × 480 Mode

Programming the 256-color, 320 × 200 mode was fairly easy. The 16-color higher resolution mode requires more effort, because of the way that the memory is organized into bit planes. To plot a single dot we need to:

1. Select and address a single bit (PEL) through the graphics address register (GAR) and bit mask register (BMR)
2. Address and set the map mask register (MMR) to 0FH and write 0 (black) to the address containing the PEL to clear the old color from the PEL
3. Set the desired PEL color through the MMR
4. Write the PEL

*The Bit Mask Register.* The bit mask register (BMR) selects the bit or bits that are modified when a byte is written to the display adapter memory. Each bit position of the display adapter memory represents a PEL in the 16-color, 640 × 480 mode. Memory from locations A0000H through A95FFH stores 38,400 bytes that represent the 307,200 PELs of this display's resolution. Memory location A0000H holds the first eight PELs, location A0001H holds the next eight PELs, and so forth. If only the left-most PEL is changed, the bit mask register is programmed with an 80H before any information is written to memory location A0000H. Likewise, other bits or multiple bits can be modified by changing the bit mask register contents.

The BMR is accessed by programming the graphics address register (GAR) with an index of 8 to select the BMR. The bit mask is then programmed. The I/O address of the GAR is 03CEH and the BMR uses I/O address 03CFH. Example 7–30 shows how the BMR is programmed so all eight bits (eight PELs) change together.

**EXAMPLE 7–30**

```
0000 BA 03CE                MOV  DX,3CEH          ;graphics address register
0003 B0 08                  MOV  AL,8             ;select bit mask register
0005 EE                     OUT  DX,AL

0006 BA 03CF                MOV  DX,03CFH         ;bit mask register
0009 B0 FF                  MOV  AL,0FFH          ;select all 8 bits
000B EE                     OUT  DX,AL
```

*The Map Mask Register.* The map mask register (MMR) selects the bit planes that are enabled for a write operation. If all bit planes are enabled, we write color F (bright white by default). The only color we cannot write in this manner is black (color 0). To write a new color to a PEL or up to eight PELs if the bit mask is FFH, we set all four bit planes (0FH) and write a 00H to memory. This clears it to the color black. Next we select the color to be written by placing the binary color number (0–F) into the MMR, and write the new PELs to memory.

Access to the MMR is provided by using index 2 for the sequence address register (port 3C4H). Once the MMR is accessed, we select bit planes by writing to the MMR at I/O port 3C5H. Example 7–31 shows how to write the first (left-most) PEL on the video screen and set it to color 2 (green). Note that the video memory location must always be read before it is changed. If the location is not read, the result is unpredictable. The read operation loads the video byte into an internal latch so it can be changed before it is rewritten to the video memory by the display adapter.

**EXAMPLE 7–31**

```
0000 B4 00              MOV  AH,0              ;set mode to 12H
0002 B0 12              MOV  AL,12H
0004 CD 10              INT  10H

0006 BA 03CE            MOV  DX,3CEH           ;graphics address register
0009 B0 08              MOV  AL,8              ;select bit mask register
000B EE                 OUT  DX,AL

000C BA 03CF            MOV  DX,3CFH           ;bit mask register
000F B0 80              MOV  AL,080H           ;select leftmost bit
0011 EE                 OUT  DX,AL

0012 BA 03C4            MOV  DX,3C4H           ;sequence address register
0015 B0 02              MOV  AL,2              ;select map mask register
0017 EE                 OUT  DX,AL

0018 BA 03C5            MOV  DX,3C5H           ;map mask register
001B B0 0F              MOV  AL,0FH            ;enable all planes
001D EE                 OUT  DX,AL
001E B8 A000            MOV  AX,0A000H         ;address video memory
0021 8E D8              MOV  DS,AX
0023 BF 0000            MOV  DI,0
0026 8A 05              MOV  AL,[DI]           ;must read first
0028 C6 05 00           MOV  BYTE PTR [DI],0   ;clear old color

002B B0 02              MOV  AL,02H            ;select color 2
002D EE                 OUT  DX,AL

002E C6 05 FF           MOV  BYTE PTR[DI],0FFH ;write memory
```

Suppose a procedure that plots any PEL on the video display is required. Example 7–32 shows such a procedure and a program that displays some dots at various points and in various colors on the video screen. Look very closely for the single red dot below and to the right of the cyan line when you execute this program.

**EXAMPLE 7–32**

```
0000                    CODE   SEGMENT 'CODE'
                               ASSUME CS:CODE

0000                    MAIN   PROC FAR

0000 B8 A000                   MOV  AX,0A000H         ;address video RAM
0003 8E D8                     MOV  DS,AX
0005 FC                        CLD                    ;select auto-increment

0006 B4 00                     MOV  AH,0              ;set mode to 12H
0008 B0 12                     MOV  AL,12H
000A CD 10                     INT  10H

000C B9 000A                   MOV  CX,10
000F BB 000A                   MOV  BX,10             ;row address
0012 BE 0064                   MOV  SI,100            ;column address
0015 B2 03                     MOV  DL,3              ;color 3 (cyan)
0017                    MAIN1:                        ;plot 10 dots
0017 E8 001B                   CALL DOT
001A 46                        INC  SI
001B E2 FA                     LOOP MAIN1
```

```
001D BB 0028                    MOV   BX,40             ;row address
0020 BE 00C8                    MOV   SI,200            ;column address
0023 B2 04                      MOV   DL,4              ;color 4 (red)
0025 E8 000D                    CALL  DOT
0028                    MAIN2:
0028 B4 06                      MOV   AH,6              ;wait for key
002A B2 FF                      MOV   DL,0FFH
002C CD 21                      INT   21H
002E 74 F8                      JZ    MAIN2

0030 B8 4C00                    MOV   AX,4C00H          ;exit to DOS
0033 CD 21                      INT   21H

0035                    MAIN    ENDP

0035                    DOT     PROC NEAR

0035 51                         PUSH  CX
0036 52                         PUSH  DX                ;save color

0037 8B C3                      MOV   AX,BX
0039 BD 0280                    MOV   BP,640            ;find video address
003C F7 E5                      MUL   BP
003E 03 C6                      ADD   AX,SI
0040 83 D2 00                   ADC   DX,0
0043 BD 0008                    MOV   BP,8
0046 F7 F5                      DIV   BP                ;get byte address
0048 8B F8                      MOV   DI,AX
004A B0 80                      MOV   AL,80H            ;form bit mask
004C 83 FA 00                   CMP   DX,0
004F 74 04                      JE    DOT1
0051 8A CA                      MOV   CL,DL
0053 D2 E8                      SHR   AL,CL             ;adjust bit mask
0055                    DOT1:
0055 50                         PUSH  AX                ;save bit mask
0056 BA 03CE                    MOV   DX,3CEH           ;graphics address register
0059 B0 08                      MOV   AL,8              ;select bit mask register
005B EE                         OUT   DX,AL

005C BA 03CF                    MOV   DX,3CFH           ;bit mask register
005F 58                         POP   AX                ;get bit mask
0060 EE                         OUT   DX,AL

0061 BA 03C4                    MOV   DX,3C4H           ;sequence address register
0064 B0 02                      MOV   AL,2              ;select map mask register
0066 EE                         OUT   DX,AL

0067 BA 03C5                    MOV   DX,3C5H           ;map mask register
006A B0 0F                      MOV   AL,0FH            ;enable all planes
006C EE                         OUT   DX,AL

006D 8A 05                      MOV   AL,[DI]           ;must read first
006F C6 05 00                   MOV   BYTE PTR [DI],0   ;clear old color

0072 58                         POP   AX                ;select color
0073 50                         PUSH  AX
0074 EE                         OUT   DX,AL

0075 C6 05 FF                   MOV   BYTE PTR[DI],0FFH ;write memory

0078 5A                         POP   DX
0079 59                         POP   CX
007A C3                         RET
```

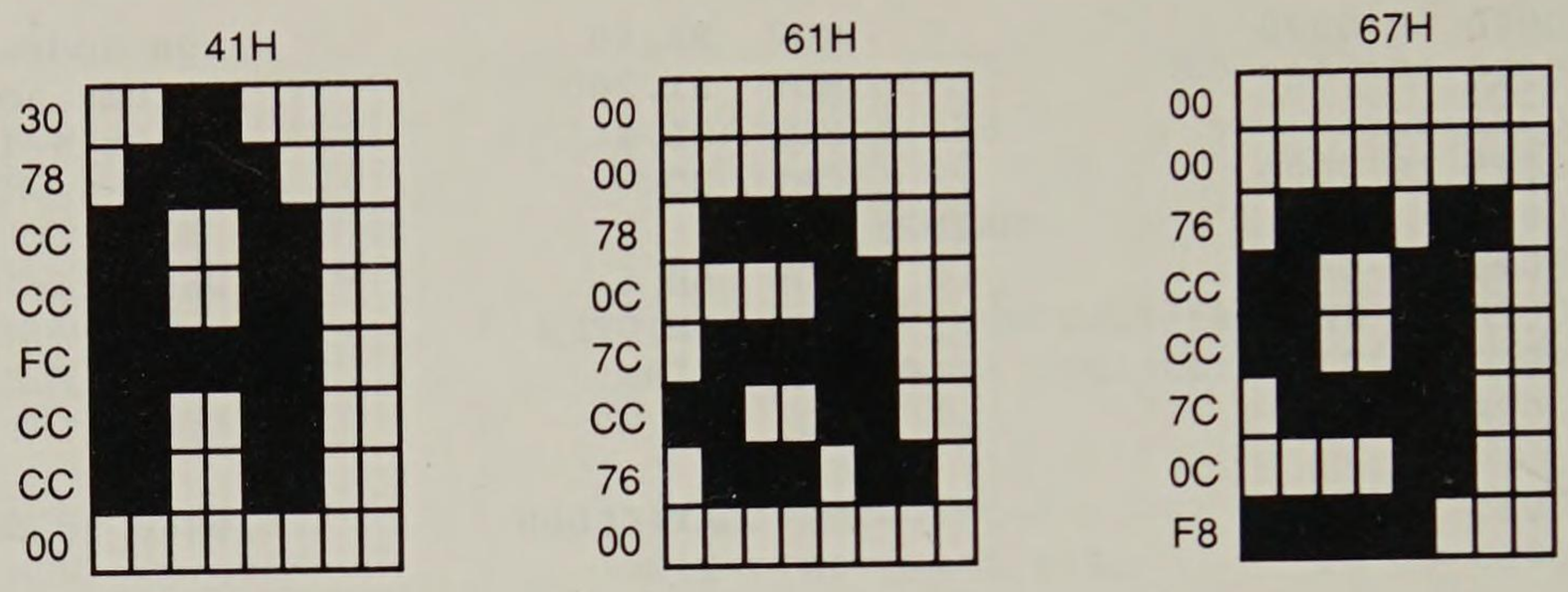



**FIGURE 7–3** A few 8 × 8 characters for the 640 × 480 VGA display.

```
007B                    DOT     ENDP

007B                    CODE    ENDS

                                END   MAIN
```

*Text with Graphics.* Displaying text in a graphics display mode such as mode 12H (16-color, 640 × 480) is also difficult. We can sometimes use the DOS function call 21H to place text on the screen, but only if we can allow video information below the text to change to both the text and its background color selected by the video attribute byte. A better way to display text in a graphics mode is to use a lookup table that contains the graphic display characters so they appear without a different background color.

The video BIOS ROM usually contains 8 × 8, 8 × 14, 9 × 14, 8 × 16, and 9 × 16 character sets. These character sets, or your own, may be used as a basis for displaying text in a graphics display mode. Figure 7–3 shows the format of the 8 × 8 character set and a few example characters. We most often use 8 × 8 characters for the 640 × 480 resolution display. The character set is obtained from the video BIOS ROM via the INT 10H function, which returns the address of the character set in ES:BP.

**EXAMPLE 7–33**

```
0000                    DATA    SEGMENT

0000 0400 [             CHAR    DB    1024 DUP (?)
             00
               ]

0400                    DATA    ENDS

0000                    CODE    SEGMENT 'CODE'
                                ASSUME CS:CODE,DS:DATA

0000                    GETC    PROC FAR

0000 FC                         CLD                     ;select increment

0001 B4 11                      MOV   AH,11H            ;get ROM character set
0003 B0 30                      MOV   AL,30H
0005 B7 03                      MOV   BH,03H            ;select 8 x 8
0007 CD 10                      INT   10H

0009 8C C0                      MOV   AX,ES             ;address memory
000B 8E D8                      MOV   DS,AX
```

```
000D B8 ---- R                  MOV  AX,DATA
0010 8E C0                      MOV  ES,AX
0012 BF 0000 R                  MOV  DI,OFFSET CHAR
0015 8B F5                      MOV  SI,BP

0017 B9 0400                    MOV  CX,1024          ;load count
001A F3/ A4                     REP MOVSB             ;copy character set

001C CB                         RET

001D                    GETC    ENDP

001D                    CODE    ENDS

                                END  GETC
```

Example 7–33 shows how the 8 × 8 standard character set is copied from the video BIOS ROM into the data segment at memory location CHAR. Because each ASCII character requires eight bytes of memory, and because the ASCII code contains 128 characters, the size of the character table is 1,024 bytes in length. Note that BH = 03H to obtain the 8 × 8 standard character set. The extended character set (code 128–255) is obtained in the same fashion, except BH = 04H.

**EXAMPLE 7–34**

```
0021                    PUTC    PROC NEAR             ;display character
0021 52                         PUSH DX
0022 56                         PUSH SI
0023 BE 0000 R                  MOV  SI,OFFSET CHAR
0026 32 E4                      XOR  AH,AH            ;adjust ASCII code
0028 D1 E0                      SHL  AX,1             ;multiply by 8
002A D1 E0                      SHL  AX,1
002C D1 E0                      SHL  AX,1
002E 03 F0                      ADD  SI,AX            ;adjust CHAR address

0030 52                         PUSH DX               ;address video byte
0031 8A C6                      MOV  AL,DH
0033 32 E4                      XOR  AH,AH
0035 B9 02D0                    MOV  CX,720
0038 F7 E1                      MUL  CX
003A 8B F8                      MOV  DI,AX
003C 5A                         POP  DX
003D 8A C2                      MOV  AL,DL
003F 32 E4                      XOR  AH,AH
0041 03 F8                      ADD  DI,AX

0043 B9 0008                    MOV  CX,8             ;set count for 8 bytes
0046                    PUTC1:
0046 BA 03CE                    MOV  DX,3CEH          ;address bit mask register
0049 B0 08                      MOV  AL,8
004B EE                         OUT  DX,AL
004C 42                         INC  DX               ;select bit mask register
004D AC                         LODSB                 ;get mask pattern
004E EE                         OUT  DX,AL

004F BA 03C4                    MOV  DX,3C4H          ;address map mask register
0052 B0 02                      MOV  AL,2
0054 EE                         OUT  DX,AL
0055 42                         INC  DX
```

```
0056 B0 0F                      MOV   AL,0FH          ;select all bit planes
0058 EE                         OUT   DX,AL
0059 26: 8A 05                  MOV   AL,ES:[DI]     ;read byte
005C 32 C0                      XOR   AL,AL          ;write black
005E AA                         STOSB
005F 4F                         DEC   DI
0060 8A C3                      MOV   AL,BL          ;get color
0062 EE                         OUT   DX,AL          ;set bit planes to color
0063 B0 FF                      MOV   AL,0FFH
0065 AA                         STOSB
0066 4F                         DEC   DI
0067 83 C7 50                   ADD   DI,80
006A E2 DA                      LOOP  PUTC1          ;repeat 8 times
006C 5E                         POP   SI
006D 5A                         POP   DX
006E C3                         RET

006F                    PUTC    ENDP
```

Once the character set is fetched from the video BIOS, it is used to display text on the graphics display. The procedure listed in Example 7–34 uses the character table stored in the data segment in Example 7–33 to display ASCII text on a graphics display. This procedure assumes the display uses nine scanning lines for each character line and 80 characters appear across the screen on a character line. This allows 53 text lines, each containing 80 characters, to be displayed on the graphics mode screen. Before the procedure is called, AL = ASCII character to be displayed, DH = character line (0–52), DL = character column (0–79), and BL = text color (0–F). This procedure does not display a background color for the text. Instead, it superimposes the text on top of whatever graphics presentation appears on the video display. This same procedure can display any character set as long as it is an $8 \times 8$.

**EXAMPLE 7–35** (pp. 244–247)

```
0000                    STACK   SEGMENT STACK

0000 0400 [                     DW    1024 DUP (?)
           0000
                 ]
0800                    STACK   ENDS

0000                    DATA    SEGMENT

0000 0400 [             CHAR    DB    1024 DUP (?)
           00
                 ]
0400 54 68 69 73 20     MES1    DB    'This is a test, only a test.',0
         69 73 20 61 20
         74 65 73 74 2C
         20 6F 6E 6C 79
         20 61 20 74 65
         73 74 2E 00
041D 54 68 69 73 20     MES2    DB    'This line is blue, above is white.',0
         6C 69 6E 65 20
         69 73 20 62 6C
         75 65 2C 20 61
         62 6F 76 65 20
         69 73 20 77 68
         69 74 65 2E 00
```

```
0440                    DATA    ENDS

0000                    CODE    SEGMENT 'CODE'
                                ASSUME CS:CODE,DS:DATA,SS:STACK

0000                    GETC    PROC NEAR

0000 FC                         CLD                     ;select increment

0001 1E                         PUSH DS
0002 06                         PUSH ES

0003 B4 11                      MOV  AH,11H             ;get ROM character set
0005 B0 30                      MOV  AL,30H
0007 B7 03                      MOV  BH,03H             ;select 8 x 8
0009 CD 10                      INT  10H

000B 8C C0                      MOV  AX,ES              ;address memory
000D 8E D8                      MOV  DS,AX
000F B8 ---- R                  MOV  AX,DATA
0012 8E C0                      MOV  ES,AX
0014 BF 0000 R                  MOV  DI,OFFSET CHAR
0017 8B F5                      MOV  SI,BP

0019 B9 0400                    MOV  CX,1024            ;load count
001C F3/ A4                     REP MOVSB               ;copy character set
001E 07                         POP  ES
001F 1F                         POP  DS
0020 C3                         RET

0021                    GETC    ENDP

0021                    PUTC    PROC NEAR               ;display character

0021 52                         PUSH DX
0022 56                         PUSH SI

0023 BE 0000 R                  MOV  SI,OFFSET CHAR

0026 32 E4                      XOR  AH,AH              ;adjust ASCII code
0028 D1 E0                      SHL  AX,1               ;multiply by 8
002A D1 E0                      SHL  AX,1
002C D1 E0                      SHL  AX,1
002E 03 F0                      ADD  SI,AX              ;adjust CHAR address
0030 52                         PUSH DX
0031 8A C6                      MOV  AL,DH
0033 32 E4                      XOR  AH,AH
0035 B9 02D0                    MOV  CX,720
0038 F7 E1                      MUL  CX
003A 8B F8                      MOV  DI,AX
003C 5A                         POP  DX
003D 8A C2                      MOV  AL,DL
003F 32 E4                      XOR  AH,AH
0041 03 F8                      ADD  DI,AX

0043 B9 0008                    MOV  CX,8               ;set count for 8 bytes
0046                    PUTC1:
0046 BA 03CE                    MOV  DX,3CEH            ;address bit mask register
0049 B0 08                      MOV  AL,8
004B EE                         OUT  DX,AL
004C 42                         INC  DX                 ;select bit mask register
004D AC                         LODSB                   ;get mask pattern
```

```
004E EE                         OUT   DX,AL

004F BA 03C4                    MOV   DX,3C4H          ;address map mask register
0052 B0 02                      MOV   AL,2
0054 EE                         OUT   DX,AL
0055 42                         INC   DX
0056 B0 0F                      MOV   AL,0FH           ;select all bit planes
0058 EE                         OUT   DX,AL
0059 26: 8A 05                  MOV   AL,ES:[DI]       ;read byte
005C 32 C0                      XOR   AL,AL            ;write black
005E AA                         STOSB
005F 4F                         DEC   DI
0060 8A C3                      MOV   AL,BL            ;get color
0062 EE                         OUT   DX,AL            ;set bit planes to color
0063 B0 FF                      MOV   AL,0FFH
0065 AA                         STOSB
0066 4F                         DEC   DI
0067 83 C7 50                   ADD   DI,80
006A E2 DA                      LOOP  PUTC1            ;repeat 8 times

006C 5E                         POP   SI
006D 5A                         POP   DX
006E C3                         RET

006F                    PUTC    ENDP

006F                    DISP_S  PROC NEAR              ;display string

006F AC                         LODSB                  ;get character
0070 0A C0                      OR    AL,AL            ;test for end of string
0072 74 07                      JE    DISP_S1
0074 E8 FFAA                    CALL  PUTC             ;display character
0077 FE C2                      INC   DL               ;address next column
0079 EB F4                      JMP   DISP_S
007B                    DISP_S1:
007B C3                         RET

007C                    DISP_S  ENDP

007C                    DISP_G  PROC NEAR              ;set display to cyan

007C B9 9600                    MOV   CX,38400         ;set count
007F BF 0000                    MOV   DI,0             ;address video
0082 BA 03CE                    MOV   DX,3CEH          ;address bit mask register
0085 B0 08                      MOV   AL,8
0087 EE                         OUT   DX,AL
0088 42                         INC   DX               ;select bit mask
0089 B0 FF                      MOV   AL,0FFH
008B EE                         OUT   DX,AL
008C BA 03C4                    MOV   DX,3C4H          ;address map mask register
008F B0 02                      MOV   AL,2
0091 EE                         OUT   DX,AL
0092 42                         INC   DX               ;select map mask register
0093                    DISP_G1:
0093 B0 0F                      MOV   AL,0FH           ;select all planes
0095 EE                         OUT   DX,AL
0096 26: 8A 05                  MOV   AL,ES:[DI]       ;read memory
0099 32 C0                      XOR   AL,AL            ;write black
009B AA                         STOSB
009C 4F                         DEC   DI
009D B0 03                      MOV   AL,3             ;select cyan
009F EE                         OUT   DX,AL
```

```
00A0 B0 FF                      MOV  AL,0FFH        ;write cyan
00A2 AA                         STOSB
00A3 E2 EE                      LOOP DISP_G1        ;repeat 38,400 times

00A5 C3                         RET

00A6                    DISP_G  ENDP

00A6                    MAIN    PROC FAR

00A6 B8 ---- R                  MOV  AX,DATA        ;address data segment
00A9 8E D8                      MOV  DS,AX

00AB B8 A000                    MOV  AX,0A000H      ;address video memory
00AE 8E C0                      MOV  ES,AX

00B0 B8 0012                    MOV  AX,0012H       ;select VGA mode 12H
00B3 CD 10                      INT  10H

00B5 E8 FF48                    CALL GETC           ;get character set
00B8 E8 FFC1                    CALL DISP_G         ;display cyan screen
00BB BE 0400 R                  MOV  SI,OFFSET MES1 ;address string
00BE B6 0A                      MOV  DH,10          ;line 10
00C0 B2 0F                      MOV  DL,15          ;column 15
00C2 B3 0F                      MOV  BL,15          ;color brightwhite
00C4 E8 FFA8                    CALL DISP_S         ;display string
00C7 BF 041D R                  MOV  DI,OFFSET MES2
00CA B6 0B                      MOV  DH,11          ;line 11
00CC B2 0F                      MOV  DL,15          ;column 15
00CE B3 09                      MOV  BL,9           ;color bright blue
00D0 E8 FF9C                    CALL DISP_S         ;display string
00D3                    MAIN1:
00D3 B4 06                      MOV  AH,06H         ;wait for key
00D5 B2 FF                      MOV  DL,0FFH
00D7 CD 21                      INT  21H
00D9 74 F8                      JZ   MAIN1

00DB B8 4C00                    MOV  AX,4C00H       ;return to DOS
00DE CD 21                      INT  21H

00E0                    MAIN    ENDP

00E0                    CODE    ENDS

                                END  MAIN
```

Example 7–35 illustrates how to display a graphics screen and place text on top of the graphics. This program selects VGA mode 12H and then displays a cyan screen. Next, it places two lines of text in two different colors on top of the cyan screen. The first line is bright white and the second line is bright blue. Experiment with this program by changing the background screen and the text to other colors.

## 7–5 DISK FILES

Data are stored on the disk in the form of files. The disk is organized in four main areas: the boot sector, the file allocation table (FAT), the root directory, and the data storage area. The

first sector on the disk is the boot sector. The *boot sector* contains a program called a *bootstrap loader* that loads the disk operating system (usually DOS) from the disk into the memory when power is applied to the computer. The FAT stores codes that indicate which disk clusters contain data, which are free, and which are bad. All references to a disk file are handled through the FAT and a directory. The root directory is where all subdirectory names and files are referenced. The disk files are all considered sequential access files, meaning that they are accessed a byte at a time from the beginning of the file toward the end.

## Disk Organization

Figure 7–4 illustrates the organization of sectors and tracks on the surface of a disk. This organization applies to both floppy and hard- or fixed-disk memory systems. The outer track is always track 0 and the inner track is 39 (double-density) or 79 (high-density) on floppy disks. The inner track on a hard disk is determined by the disk size and could be 10000 or higher for very large hard disks.

Floppy disks come in two different sizes: the 5¼″ mini-floppy and the 3½″ micro-floppy. The 5¼″ mini-floppy disk is currently available as a double-sided, double-density (DSDD) disk that stores 360K bytes and a high-density (HD) disk that stores 1.2M bytes. The 3½″ micro-floppy is available as a double-sided, double-density (DSDD) disk that stores 720K bytes, a high-density (HD) disk that stores 1.44M bytes, and an extra-high-density (EHD or ED) disk that stores 2.88M bytes. In all cases, each type of disk memory organizes its surface into *tracks* that contain *sectors*. Each sector on a disk usually contains 512

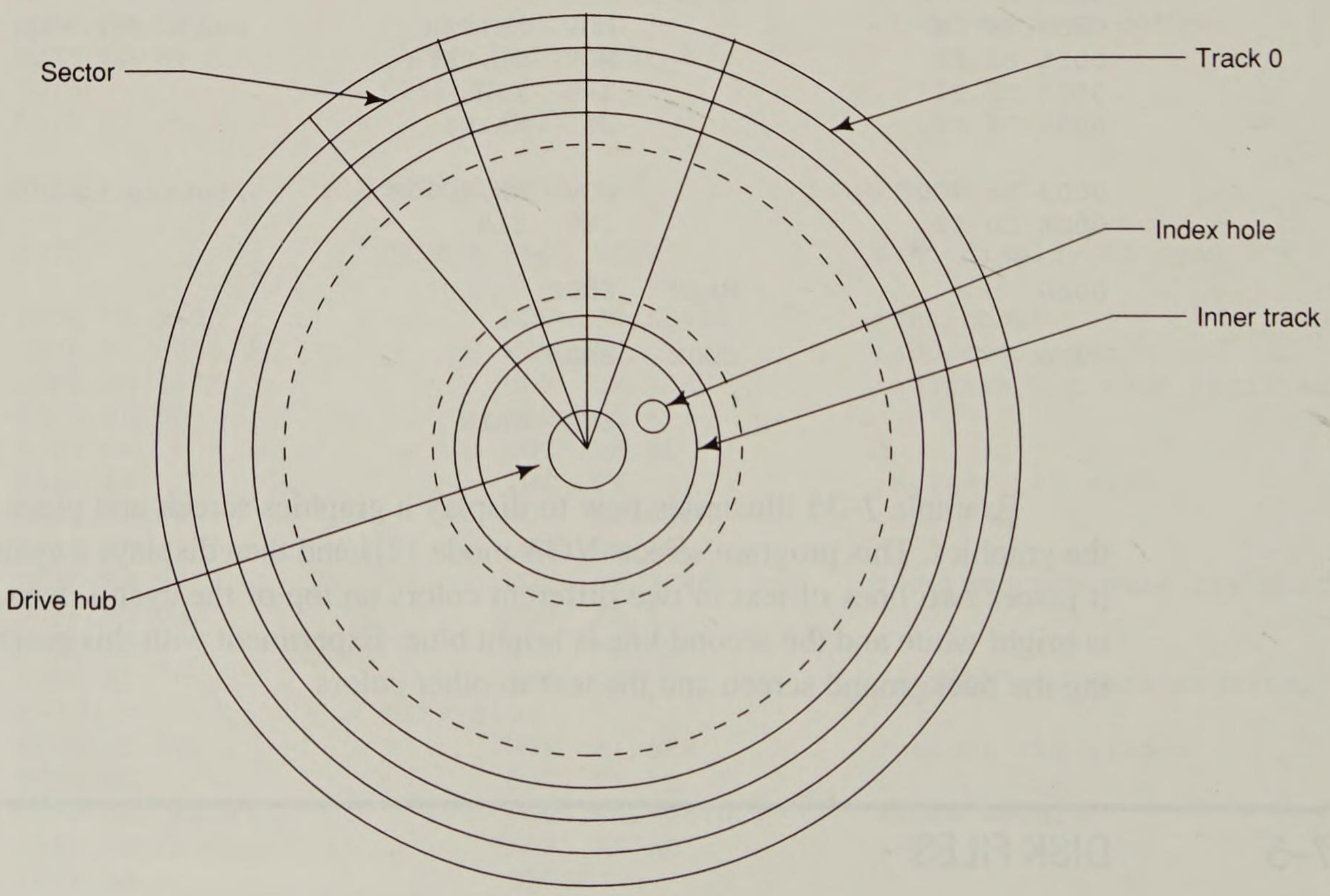



**FIGURE 7–4** The format of a 5¼″ floppy disk.

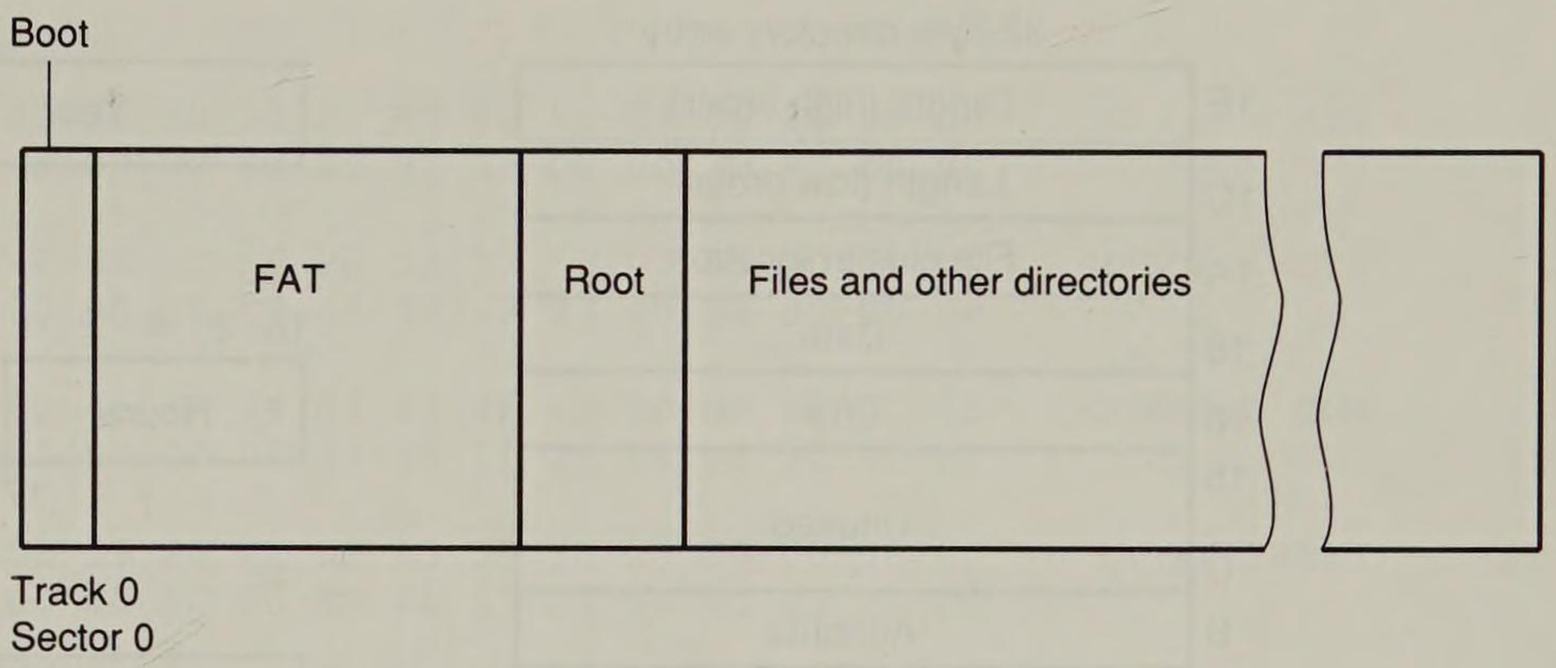



**FIGURE 7–5** The main data storage areas on a disk.

bytes of data. The number of sectors and tracks varies from one form to another. For example, the 5¼″ double-density, double-sided disk contains 40 tracks with 9 sectors per track ($40 \times 9 \times 512 = 180$K bytes) per side. A pair of tracks (top and bottom) is called a *cylinder*. We often use the term *cylinder* in place of *track* and *upper head* or *lower head* in place of *side*. Some fixed-disk drives have up to 16 heads per cylinder, because they contain more than one disk platter.

Figure 7–5 shows the organization of data on a disk. The length of the FAT is determined by the size of the disk, because each disk sector or group of sectors is represented in the FAT by a 16-bit number. Note that some extremely large fixed disks use a 32-bit number as an entry in the FAT. Another variation is a 12-bit FAT. The length of the root directory is fixed at 4K bytes on a floppy disk (larger on a hard disk), but never more than one cylinder. The boot sector is always a single *512-byte*-long sector located in the outer track (track 0) at sector 0, the first sector.

The boot sector contains a bootstrap loader program, which is read into RAM when the system is powered. After the bootstrap loader is read into RAM, it executes and loads the IO.SYS* and MSDOS.SYS† programs into RAM. Next, the bootstrap loader passes control to the MSDOS control program and the computer is now under the control of the DOS command processor COMMAND.COM. This is where the CONFIG.SYS file is used to configure the system and where the AUTOEXEC.BAT file is executed.

The FAT indicates which sectors or clusters of sectors are free, which are corrupted (unusable), and which contain data. The FAT table is referenced each time DOS writes data to the disk, so that it can find a free sector. Each free cluster is indicated by a 0000H in the FAT and each occupied cluster is indicated by the cluster number. A *cluster* can be one to any number of sectors in length. Many fixed-disk memory systems use four or more sectors per cluster, which means the smallest file allocation unit is $512 \times 4$ or 2,048 bytes in length if a cluster contains four sectors. The larger the hard-disk drive, the more sectors in a cluster. A 16-bit FAT table accesses 64K—17 clusters. A 0000H code indicates a free cluster, FFF7H indicates a bad cluster, and the remaining cluster numbers (0001H–FFF0H) are available. If a cluster is four sectors of 512 bytes, a fixed disk containing a 16-bit FAT accesses a maximum of 65,519 clusters. Because a sector is 512 bytes and a cluster is 2,048 bytes, or four sectors, this hard-disk drive contains a maximum of 134,182,912 bytes of

---

*IO.SYS is an I/O control program provided by Microsoft Corporation with Microsoft DOS.

†MSDOS.SYS is the Microsoft Disk Operating System.

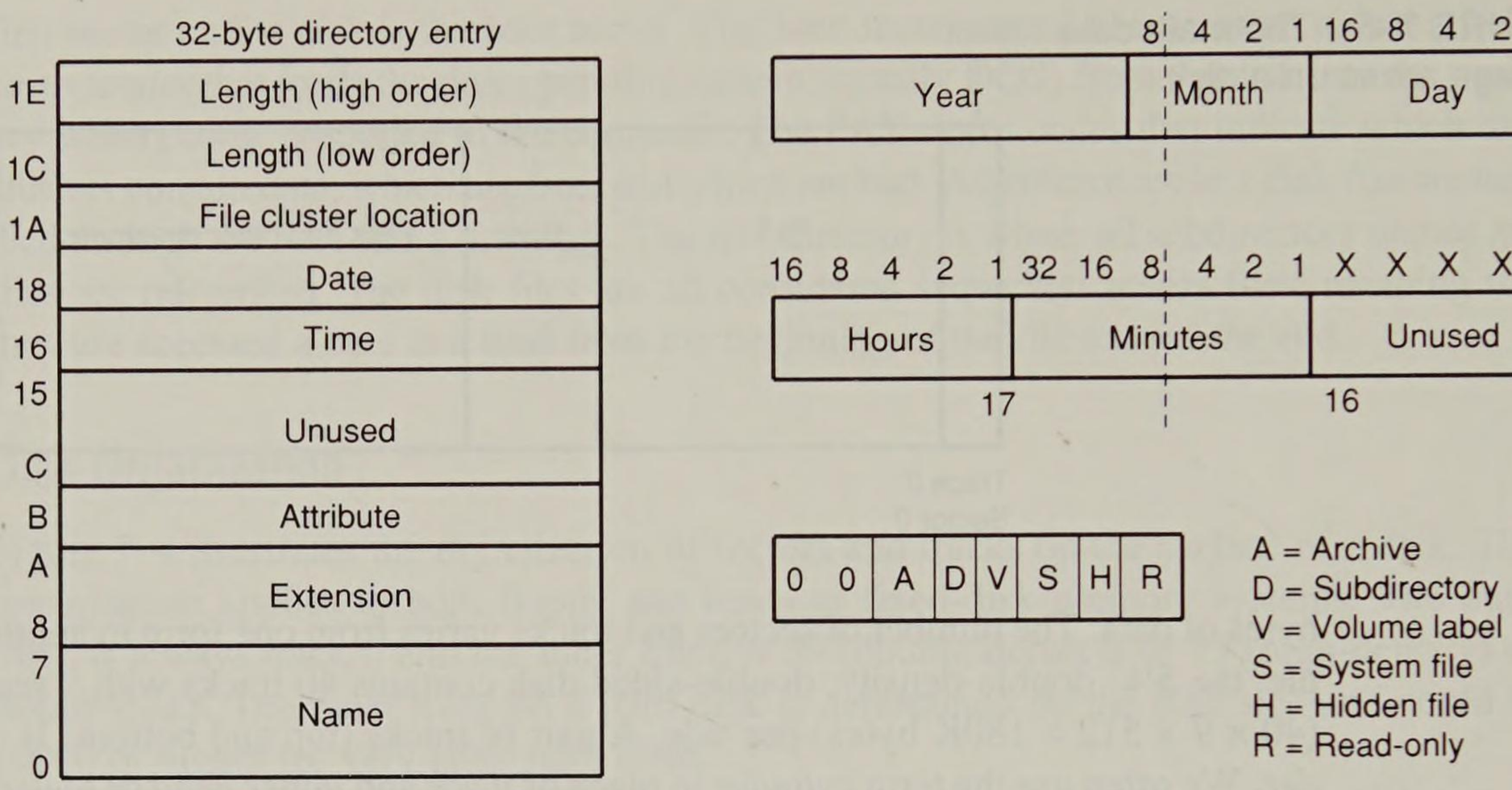



**FIGURE 7–6** The format for any directory or subdirectory entry.

data. If a cluster contains eight sectors (4,096 bytes), then the maximum capacity of the hard-disk drive is 268,365,824 bytes. If a larger hard disk exists, either more sectors per clusters are allocated or the FAT is increased to 24 or 32 bits.

Figure 7–6 shows the format of each directory entry in the root or any other directory or subdirectory. Each entry contains the name, extension, attribute, time, date, location, and length. The length of the file is stored as a 32-bit number. This means that a file can have a maximum length of 4G bytes. The location is the starting cluster number. The root directory on a floppy disk has room for 128 of these 32-byte directory entries. On a hard-disk drive the root directory contains 256 or more entries. When the root directory is full, the disk is full. This is why it is important to use subdirectories, which can contain any number of files desired. A single subdirectory requires one entry in the root directory. A sub-subdirectory entry is stored in the subdirectory, not the root directory. This allows an almost infinite combination of files stored in subdirectories.

The location in the directory entry holds the first cluster number where the file is stored on the disk. The FAT location that pertains to this cluster location holds either the next cluster number, where the file continues, or an FFFFH, to indicate the end of the file. This is required so DOS can store files on the disk in fragments (pieces) if required. Fragments cause problems after a disk is in use for a long period of time. As programs and files are stored and erased, they seem to occupy a larger number of smaller and smaller fragments of disk space. Eventually, a file that requires four clusters will be stored in four clusters located at various locations throughout the surface of the disk. This significantly increases access times for large files. This problem is solved by using a de-frag program such as PCTOOLS* compress or optimize. It is recommended that the hard disk be de-fragmented occasionally, especially if access time becomes a noticeable problem.

---

*PCTOOLS is registered to Central Point Software.

**EXAMPLE 7–36**

```
0000 49 4F 20 20 20 20 20 20 53 59 53 07 00 00 00 00    IO      SYS
0010 00 00 00 00 00 00 00 00 93 11 02 00 39 82 00 00

0020 4D 53 44 4F 53 20 20 20 53 59 53 07 00 00 00 00    MSDOS   SYS
0030 00 00 00 00 00 00 C0 44 93 12 13 00 92 00 00 00

0040 43 4F 4D 4D 41 4E 44 20 43 4F 4D 00 00 00 00 00    COMMAND COM
0050 00 00 00 00 00 00 00 00 93 11 26 00 B5 92 00 00

0060 42 41 52 52 59 20 42 52 45 59 20 28 00 00 00 00    BARRY BREY
0070 00 00 00 00 00 00 E0 AD 6A 13 00 00 00 00 00 00

0080 50 43 54 4F 4F 4C 53 20 20 20 20 10 00 00 00 00    PCTOOLS
0090 00 00 00 00 00 00 80 AE 6A 13 5C 00 00 00 00 00

00A0 44 4F 53 20 20 20 20 20 20 20 20 10 00 00 00 00    DOS
00B0 00 00 00 00 00 00 E0 B0 6A 13 4E 00 00 00 00 00

00C0 52 55 4E 5F 46 57 20 20 42 41 54 00 00 00 00 00    FUN_FW  BAT
00D0 00 00 00 00 00 00 40 BD 6A 13 97 0F 4A 00 00 00

00E0 46 4F 4E 54 57 41 52 45 20 20 20 10 00 00 00 00    FONTWARE
00F0 00 00 00 00 00 00 60 BD 6A 13 6E 00 00 00 00 00
```

Example 7–36 shows how part of the root directory appears in a hexadecimal dump. Try to identify the date, time, location, and length of each entry. Also identify the attribute for each entry. The listing shows hexadecimal data and ASCII data, as is customary in most computer dumps.

Files are usually accessed through DOS INT 21H function calls. There are two ways to access a file using an INT 21H. One method uses a file control block (FCB) and the other uses a file handle. Today, all softwares access a file using a file handle, so this text uses file handles for file access. File control blocks are a carryover from an earlier operating system called CP/M* (control program/microprocessor), which was used with 8-bit computer systems based on the Z80, 8080, or 8085 microprocessor. All DOS INT 21H function calls that use file control blocks should be avoided in new software.

## Sequential Access Files

All DOS files are sequential access files. A sequential access file is stored and addressed from the beginning toward its end. This means that the first byte, and all bytes between it and the last, must be accessed to read the last byte. Fortunately, files are read and written with the DOS INT 21H function calls (refer to Appendix A), which simplifies file access and data manipulation. This section of the text describes how to create, read, write, delete, and rename a sequential access file using the DOS INT 21H function calls.

## File Creation

Before a file is used, it must exist on the disk. A file is *created* by the INT 21H function call number 3CH. To create a file, the name of the file and its extension must be stored in

---

*CP/M is a registered trademark of Digital Research Corporation.

memory at a location addressed by DS:DX before calling the function. The CX register must also contain the attribute of the file created by function 3CH. Care must be taken in naming the file, because if another file exists with the same name, create will erase the old file and create a new one without any data. Note that subdirectories can also be created with function 3CH.

**EXAMPLE 7–37**

```
0000                    DAT     SEGMENT
0000 44 4F 47 2E 54     FILE1   DB    'DOG.TXT',0               ;file name DOG.TXT
       58 54 00
0008 43 3A 44 41 54     FILE2   DB    'C:DATA.DOC',0            ;file C:DATA.DOC
       41 2E 44 4F 43
       00
0013 43 3A 5C 44 52     FILE3   DB    'C:\DREAD\ERROR.FIL',0    ;file C:\DREAD\ERROR.FIL
       45 41 44 5C 45
       52 52 4F 52 2E
       46 49 4C 00

0026                    DAT     ENDS
```

A *file name* is always stored as an ASCII-Z string and may contain the disk drive letter and directory path(s) if needed. Example 7–37 shows several ASCII-Z string file names stored in a data segment for access by the file utilities. An *ASCII-Z string* is a character string that ends with a 00H or null character.

Suppose you have filled a 256-byte memory buffer area with data to be stored in a new file called DATA.NEW on the default disk drive. Before data can be written to this new file, the file must be created. Example 7–38 lists a short procedure that creates this new file on the disk.

**EXAMPLE 7–38**

```
0000                    DAT     SEGMENT

0000 44 41 54 41 2E     FILE1   DB    'DATA.NEW',0      ;file name DATA.NEW
       4E 45 57 00
0009 0100[              BUFFER  DB    256 DUP (?)       ;data buffer
         ??
            ]

0109                    DAT     ENDS

0000                    COD     SEGMENT 'CODE'
                                ASSUME  CS:COD,DS:DAT

0000 B8 ---- R                  MOV   AX,DAT
0003 8E D8                      MOV   DS,AX             ;load DS

0005 B4 3C                      MOV   AH,3CH            ;load create function
0007 33 C9                      XOR   CX,CX             ;00H = attribute
0009 BA 0000 R                  MOV   DX,OFFSET FILE1   ;address ASCII-Z name
000C CD 21                      INT   21H               ;create DATA.NEW
```

**TABLE 7–4** File attribute definitions

| *Bit Position* | *Attribute* | *Function* |
|---|---|---|
| 0 | Read-only | A read-only file or subdirectory |
| 1 | Hidden | Prevents the file or subdirectory name from appearing in the directory when a DIR is used from the DOS command line |
| 2 | System | Specifies a file as a system file |
| 3 | Volume | Specifies the disk volume label |
| 4 | Subdirectory | Specifies a subdirectory name |
| 5 | Archive | Indicates that a file has been changed and should be archived |

```
000E 72 20                      JC    ERROR             ;on creation error
                                 .     .
                                 .     .
                                 .     .
0030                    COD     ENDS

                                END
```

Whenever a file is created, the CX register must contain the attributes or characteristics of the file. Table 7–4 lists the attribute bit positions and defines them. A logic one in a bit selects the attribute, while a logic zero does not. For example, to create a hidden archive file, CX is loaded with a 0022H. Note that the hidden and archives bits are set.

After returning from the INT 21H, the carry flag indicates whether an error occurred (C = 1) during the creation of the file. Some errors that occur during file creation are: path not found, no file handles available, or media error. These errors can be obtained, if needed, by INT 21H function call number 59H. If carry is cleared (C = 0), no error occurred during file creation, and the AX register contains a file handle. The *file handle* is a number used to refer to the file after it is created or opened. The file handle allows a file to be accessed without using the ASCII–Z string name of the file, speeding the operation.

## Writing to a File

Now that we have created a new file in Example 7–38, called FILE.NEW, data can be written into the file. Before writing to a file it must be created or opened. When a file is created or opened, the file handle returns in the AX register. The file handle is used to refer to the file whenever data are written or whenever the file is referred to in the future for any reason.

Function number 40H is used to write data to an opened or newly created file. In addition to loading a 40H into AH, we must load BX with the file handle, CX with the number of bytes to be written, and DS:DX with the address of the memory buffer area to be written to the disk.

**EXAMPLE 7–39**

```
                              .     .
                              .     .
                              .     .
0010 8B D8                    MOV   BX,AX               ;move handle to BX
0012 B4 40                    MOV   AH,40H              ;load write function
0014 B9 0100                  MOV   CX,256              ;load count
0017 BA 0009 R                MOV   DX,OFFSET BUFFER    ;address BUFFER
001A CD 21                    INT   21H                 ;write 256 bytes from BUFFER

001C 72 32                    JC    ERROR1              ;on write error
                              .     .
                              .     .
                              .     .
```

Suppose that we must write 256 bytes from data segment memory area BUFFER to a file. This is accomplished as illustrated in Example 7–39 by using function 40H. If an error occurs during a write operation, the carry flag is set. If no error occurs, the carry flag is cleared and the number of bytes written to the file are returned in the AX register. Errors that occur during write operations usually indicate that the disk is full or that there is some type of media error. Media errors are caused by a bad disk (floppy) or a problem with the disk electronics.

## Opening, Reading, and Closing a File

To read or write an existing file, it must be opened first. When a file is opened, DOS checks the directory to determine whether the file exists and returns the DOS file handle in register AX. The DOS file handle must be used for reading, writing, and closing a file.

**EXAMPLE 7–40**

```
0000                    READ    PROC NEAR

0000 B4 3D                      MOV   AH,3DH              ;load open function
0002 B0 00                      MOV   AL,0                ;select read
0004 BA 0000 R                  MOV   DX,OFFSET FILE1     ;address file name
0007 CD 21                      INT   21H                 ;open file

0009 72 15                      JC    ERROR               ;on error

000B 8B D8                      MOV   BX,AX               ;move file handle to BX

000D B4 3F                      MOV   AH,3FH              ;load read function
000F B9 0100                    MOV   CX,256              ;number of bytes
0012 BA 0009 R                  MOV   DX,OFFSET BUFFER    ;address BUFFER
0015 CD 21                      INT   21H                 ;read 256 bytes

0017 72 07                      JC    ERROR               ;on error

0019 B4 3E                      MOV   AH,3EH              ;load close function
001B CD 21                      INT   21H                 ;close file

001D 72 01                      JC    ERROR               ;on error

001F C3                         RET

                        READ    ENDP
```

Example 7–40 shows a sequence of instructions that open a file, read 256 bytes from the file into memory area BUFFER, and then close the file. When a file is opened (AH = 3DH), the AL register specifies the type of operation allowed for the opened file. If AL = 00H, the file is opened for a read operation; if AL = 01H, the file is opened for a write operation; and if AL = 02H, the file is opened for a read or a write operation.

Function number 3FH causes a file to be read. As with the write function, BX contains the file handle, CX contains the number of bytes to be read, and DS:DX contains the location of a memory area where the data are stored. As with all disk functions, the carry flag indicates an error when C = 1. If C = 0, the AX register indicates the number of bytes actually read from the file.

Closing a file is very important. If a file is left open, some serious problems occur that can actually destroy the disk and all its data. If a file is written and not closed, the FAT can become corrupted, making it difficult or impossible to retrieve data from the disk. Always be certain to close the file after it is read or written. If you suspect that a file has been written without closing, you should execute the DOS utility program CHKDSK (check disk) before any subsequent write to the disk. The CHKDSK program tests the disk and looks for lost file chains. If it detects a lost file chain (usually caused by forgetting to close a file), it can correct the problem. Run the CHKDSK /F command at the DOS prompt to fix any lost file chains.

## The File Pointer

Whenever a file is opened, written, or read, the file pointer addresses the current location in the sequential file. When a file is opened, the file pointer always addresses the first byte of the file. If a file is 1,024 bytes in length, and a read function (3FH) reads 1,023 bytes, the file pointer addresses the last byte of the file, but not the end of the file.

The *file pointer* is a 32-bit number that addresses any byte in a file. Once a file is opened, the file pointer can be changed with the move file pointer function number 42H to access any location in the file. A file pointer can be moved from the start of the file (AL = 00H), from the current location (AL = 01H), or from the end of the file (AL = 02H). In practice all three directions are used to access different parts of the file. The distance moved by the file pointer is specified by registers CX and DX. The DX register holds the least significant part of the distance and CX holds the most significant part of the distance. Register BX must contain the file handle before using function 42H to move the file pointer.

Suppose that a file exists on the disk and that you must append the file with 256 bytes of new information. When the file is opened, the file pointer addresses the first byte of the file. If you attempt to write new data without moving the file pointer to the end of the file, the new data will overwrite the first 256 bytes of the file. Example 7–41 shows a procedure that opens a file, moves the file pointer to the end of the file, writes 256 bytes of data, and then closes the file. This *appends* the file with 256 new bytes of data.

**EXAMPLE 7–41**

```
0000                    APPEND PROC NEAR

0000 B4 3D                     MOV  AH,3DH              ;load open function
0002 B0 01                     MOV  AL,1                ;select write
```

```
0004 BA 0000 R                  MOV   DX,OFFSET FILE1    ;address file name
0007 CD 21                      INT   21H                ;open file

0009 72 21                      JC    ERROR              ;on error

000B 8B D8                      MOV   BX,AX              ;move file handle to BX

000D B4 42                      MOV   AH,42H             ;load move file pointer function
000F B0 02                      MOV   AL,02H             ;move from end
0011 33 C9                      XOR   CX,CX              ;move 0 bytes from end
0013 33 D2                      XOR   DX,DX
0015 CD 21                      INT   21H                ;move pointer to end

0017 72 13                      JC    ERROR              ;on error

0019 B4 40                      MOV   AH,40H             ;load write function
001B B9 0100                    MOV   CX,256             ;number of bytes
001E BA 0009 R                  MOV   DX,OFFSET BUFFER   ;address BUFFER
0021 CD 21                      INT   21H                ;write 256 bytes

0023 72 07                      JC    ERROR              ;on error

0025 B4 3E                      MOV   AH,3EH             ;load close function
0027 CD 21                      INT   21H                ;close file

0029 72 01                      JC    ERROR              ;on error

002B C3                         RET

                        APPEND ENDP
```

One of the more difficult file maneuvers is inserting new data into the middle of a file. Figure 7–7 shows how this is accomplished by creating a second temporary file. Notice that the part of the file before the insertion point is copied into the new file. This is

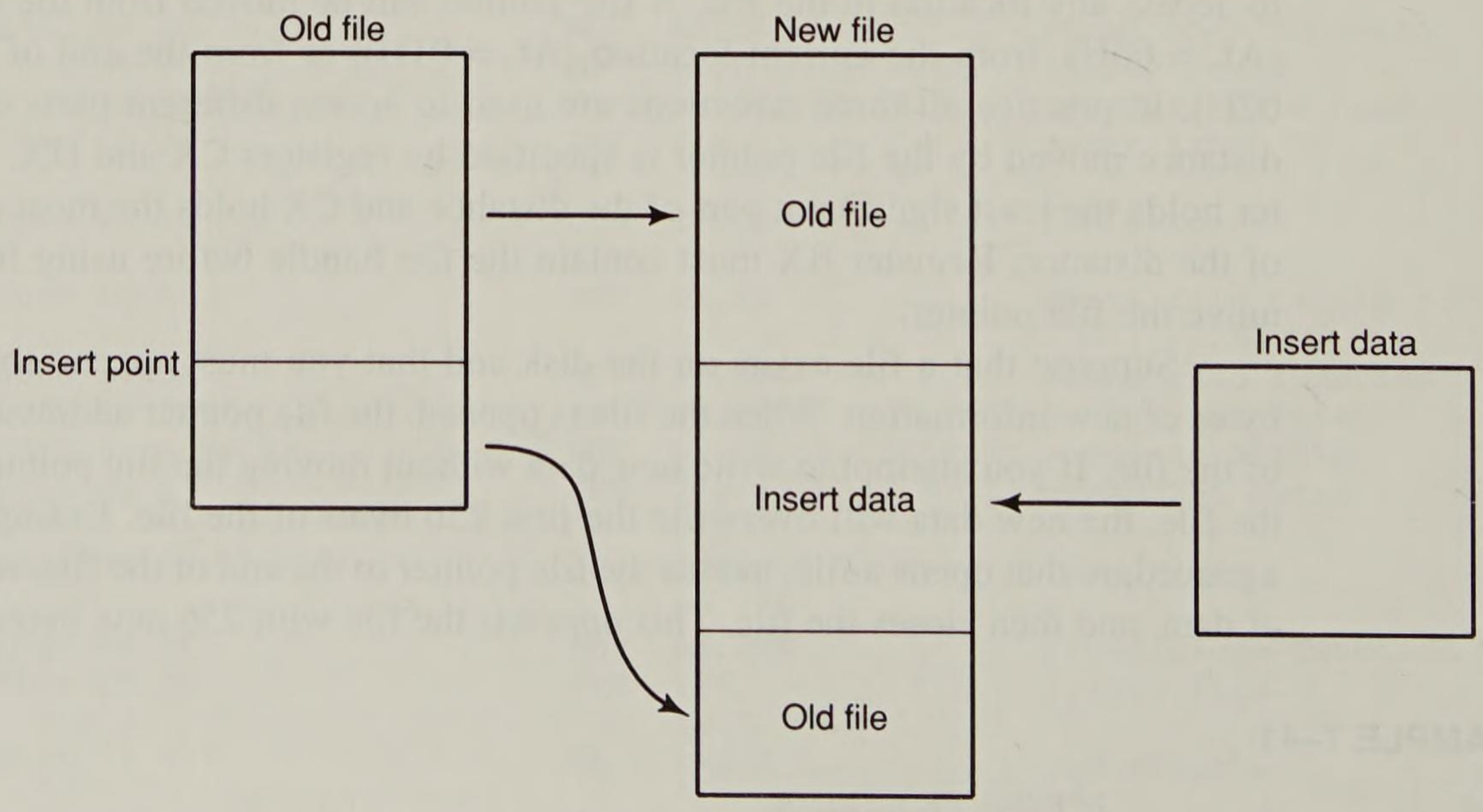



**FIGURE 7–7** Inserting new data within an old file.

followed by the new information. Then, the remainder of the file is appended to the end of the new file. Once the new file is complete, the old file is deleted and the new file is renamed to the old file name.

**EXAMPLE 7–42** (pp. 257–259)

```
0000                    DATA    SEGMENT  PUBLIC

0000 0100[              BUFFER DB    256 DUP (?)      ;buffer
          ??
             ]
0100 0000               TEMP    DW    ?               ;temporary data
0102 0000               TEMP1   DW    ?               ;temporary data
0104 0000               O_HAN   DW    ?               ;old handle
0106 0000               N_HAN   DW    ?               ;temp handle
0108 54 45 4D 50 2E     TEMPS   DB    'TEMP.$$$',0    ;temp file
        24 24 24 00

0111                    DATA    ENDS

0000                    CODE    SEGMENT 'CODE'

                                ASSUME  CS:CODE,DS:DATA

0000                    INSERT PROC FAR

0000 89 16 0100 R               MOV   TEMP,DX         ;save old file address
0004 89 0E 0102 R               MOV   TEMP1,CX        ;save count

0008 B8 3D02                    MOV   AX,3D02H        ;open old file
000B CD 21                      INT   21H
000D A3 0104 R                  MOV   O_HAN,AX        ;save old file handle
0010 73 05                      JNC   INSERT1         ;if no error

0012 E8 009F R                  CALL C_OLD            ;close old file
0015 F9                         STC                   ;indicate error
0016 CB                         RET
0017                    INSERT1:
0017 B4 3C                      MOV   AH,3CH          ;create temp file
0019 33 C9                      XOR   CX,CX
001B BA 0108 R                  MOV   DX,OFFSET TEMPS
001E CD 21                      INT   21H
0020 A3 0106 R                  MOV   N_HAN,AX        ;save temp file handle
0023 73 08                      JNC   INSERT2
0025                    EXIT:
0025 E8 00A8 R                  CALL C_TEMP           ;close temp file
0028 E8 009F R                  CALL C_OLD            ;close old file
002B F9                         STC                   ;indicate error
002C CB                         RET
002D                    INSERT2:
002D 81 FF 0100                 CMP   DI,256          ;test insert point
0031 77 07                      JA    INSERT3         ;if greater than 256
0033 0B ED                      OR    BP,BP
0035 75 03                      JNE   INSERT3         ;if greater than 256
0037 EB 17 90                   JMP   INSERT4         ;if less than 256
003A                    INSERT3:
003A B9 0100                    MOV   CX,256
003D E8 00B1 R                  CALL R_OLD            ;read old file
0040 72 E3                      JC    EXIT            ;on error
0042 E8 00BD R                  CALL W_TEMP           ;write temp file
```

```
0045 72 DE                      JC    EXIT              ;on error
0047 81 EF 0100                 SUB   DI,256            ;decrement BP-DI by 256
004B 83 DD 00                   SBB   BP,0
004E EB DD                      JMP   INSERT2
0050                    INSERT4:
0050 8B CF                      MOV   CX,DI             ;get count
0052 E8 00B1 R                  CALL  R_OLD             ;read old file
0055 72 CE                      JC    EXIT              ;on error
0057 E8 00BD R                  CALL  W_TEMP            ;write temp file
005A 72 C9                      JC    EXIT              ;on error

005C 8B 0E 0102 R               MOV   CX,TEMP1          ;get insert count
0060 1E                         PUSH  DS                ;save data segment
0061 8C C0                      MOV   AX,ES
0063 8E D8                      MOV   DS,AX             ;load DS with ES

0065 8B D6                      MOV   DX,SI             ;get address
0067 B4 40                      MOV   AH,40H            ;write insert data
0069 CD 21                      INT   21H
006B 1F                         POP   DS                ;restore DS
006C 72 B7                      JC    EXIT              ;on error
006E                    INSERT5:
006E B9 0100                    MOV   CX,256            ;write remainder of file
0071 E8 00B1 R                  CALL  R_OLD             ;read old file
0074 72 AF                      JC    EXIT              ;on error
0076 0B C0                      OR    AX,AX             ;test for end of file
0078 74 05                      JE    INSERT6           ;if end file
007A E8 00BD R                  CALL  W_TEMP            ;write temp file
007D EB EF                      JMP   INSERT5           ;repeat until end
007F                    INSERT6:
007F E8 009F R                  CALL  C_OLD             ;close old file
0082 E8 00A8 R                  CALL  C_TEMP            ;close temp file
0085 8B 16 0100 R               MOV   DX,TEMP           ;delete old file
0089 B4 41                      MOV   AH,41H
008B CD 21                      INT   21H

008D 06                         PUSH  ES                ;save ES
008E 8C D8                      MOV   AX,DS
0090 8E C0                      MOV   ES,AX             ;load ES with DS

0092 8B 3E 0100 R               MOV   DI,TEMP           ;get old file name
0096 BA 0108 R                  MOV   DX,OFFSET TEMPS
0099 B4 56                      MOV   AH,56H            ;rename file
009B CD 21                      INT   21H
009D 07                         POP   ES
009E CB                         RET

009F                    INSERT  ENDP

009F                    C_OLD   PROC NEAR

009F 8B 1E 0104 R               MOV   BX,O_HAN          ;close old file
00A3 B4 3E                      MOV   AH,3EH
00A5 CD 21                      INT   21H
00A7 C3                         RET

00A8                    C_OLD   ENDP

00A8                    C_TEMP  PROC NEAR               ;close temp file

00A8 8B 1E 0106 R               MOV   BX,N_HAN
```

```
00AC B4 3E                      MOV  AH,3EH
00AE CD 21                      INT  21H
00B0 C3                         RET

00B1                    C_TEMP  ENDP

00B1                    R_OLD   PROC NEAR

00B1 8B 1E 0104 R               MOV  BX,O_HAN
00B5 BA 0000 R                  MOV  DX,OFFSET BUFFER
00B8 B4 3F                      MOV  AH,3FH
00BA CD 21                      INT  21H
00BC C3                         RET

00BD                    R_OLD   ENDP

00BD                    W_TEMP  PROC NEAR

00BD 8B C8                      MOV  CX,AX
00BF 8B 1E 0106 R               MOV  BX,N_HAN
00C3 BA 0000 R                  MOV  DX,OFFSET BUFFER
00C6 B4 40                      MOV  AH,40H
00C8 CD 21                      INT  21H
00CA C3                         RET

00CB                    W_TEMP  ENDP

00CB                    CODE    ENDS

                        END
```

Example 7–42 shows a procedure that inserts new data, up to 64K bytes in length, into an old file. This procedure requires some input parameters to make it a general-purpose procedure. The first parameter is the old file name that is passed to the procedure through the DS:DX register. The second parameter (register CX) is the number of new bytes of information to write to the new file. The third parameter is the location (ES:SI) of the new data to be inserted in the file. The final parameter is the insertion point, or byte count of the insertion point, located in registers BP and DI, where BP contains the most-significant insertion point and DI contains the least-significant insertion point.

This procedure uses two new INT 21H function calls. The delete and rename function calls are used to delete the old file before the temporary file is renamed to the old file name. These functions, and all other DOS file functions, are listed in Appendix A.

## Random Access Files

Random access files are developed through software using DOS sequential access files. A random access file is addressed by a record number, rather than by passing through the file searching for data. The move pointer function is very important when random access files are created. Random access files are much easier to use for large volumes of data than sequential access files.

*Creating a Random Access File.* Planning is paramount in creating a random access file system. Suppose a random access file is required for storing the names of customers. Each customer record requires 16 bytes for the last name, 16 bytes for the first name, and 1 byte

for the middle initial. Each customer record contains two street address lines of 32 bytes each, a city line of 16 bytes, 2 bytes for the state code, and 9 bytes for the ZIP Code. The basic customer information alone requires 105 bytes. Additional information expands the record to 256 bytes in length. Because the business is growing, provisions are made for 5,000 customers. This means that the total required random access file is 1,280,000 bytes in length.

**EXAMPLE 7–43**

```
0000                    DATA    SEGMENT

0000 0100[              BUFFER DB   256 DUP (0)       ;buffer of 00H
          00
             ]
0100 41 3A 43 55 53     FILE   DB   'A:CUST.FIL',0    ;file name
       54 4E 46 49 4C
       00

010B                    DATA   ENDS

0000                    CODE   SEGMENT 'CODE'
                               ASSUME  CS:CODE,DS:DATA

0000                    MAKE   PROC FAR

0000 B8 ---- R                 MOV  AX,DATA          ;load DS
0003 8E D8                     MOV  DS,AX

0005 B4 3C                     MOV  AH,3CH           ;create CUST.FIL
0007 33 C9                     XOR  CX,CX
0009 BA 0100 R                 MOV  DX,OFFSET FILE
000C CD 21                     INT  21H

000E 8B D8                     MOV  BX,AX            ;handle to BX

0010 BF 1388                   MOV  DI,5000          ;load record count

0013                    MAKE1:

0013 B4 40                     MOV  AH,40H           ;write a record of 00H
0015 B9 0100                   MOV  CX,256
0018 BA 0000 R                 MOV  DX,OFFSET BUFFER
001B CD 21                     INT  21H
001D 4F                        DEC  DI
001E 75 F3                     JNZ  MAKE1            ;repeat 5,000 times

0020 B4 3E                     MOV  AH,3EH           ;close file
0022 CD 21                     INT  21H

0024 B4 4C                     MOV  AH,4CH           ;exit to DOS
0026 CD 21                     INT  21H

0028                    MAKE   ENDP

0028                    CODE   ENDS

                               END  MAKE
```

Example 7–43 illustrates a short program that creates a file called CUST.FIL and inserts 5,000 blank records of 256 bytes each. A blank record contains 00H in all bytes.

This appears to be a large file, but it fits on a single, high-density 5¼″ or 3½″ floppy disk drive. This program assumes that the disk is in drive A. Note that this program takes a considerable amount of time to execute, because it writes data to virtually every byte on the floppy disk.

*Reading and Writing a Record.* Whenever a record is read, the record number is loaded into the BP register and the procedure listed in Example 7–44 is called. This procedure assumes that FIL contains the file handle number and that the CUST.FIL remains open at all times.

**EXAMPLE 7–44**

```
0000                    READ    PROC FAR

0000 8B 1E 0100 R               MOV  BX,FIL             ;get handle
0004 B8 0100                    MOV  AX,256             ;multiply by 256
0007 F7 E5                      MUL  BP
0009 8B CA                      MOV  CX,DX
000B 8B D0                      MOV  DX,AX
000D B8 4200                    MOV  AX,4200H           ;move pointer
0010 CD 21                      INT  21H

0012 B4 3F                      MOV  AH,3FH             ;read record
0014 B9 0100                    MOV  CX,256
0017 BA 0000 R                  MOV  DX,OFFSET BUFFER
001A CD 21                      INT  21H
001C CB                         RET

001D                    READ    ENDP
```

Notice how the record number is multiplied by 256 to obtain a count for the move pointer function. In each case, the file pointer is moved from the start of the file to the desired record before it is read into memory area BUFFER. Writing a record is performed the same way as reading a record.

## Creating a Simple Database

Although many good database programs are available, we can use assembly language and a random access file to create a tailored and highly efficient database. This section shows a simple database system that catalogs videotapes for a small, home videotape library. Before the program is created, the size of each record is determined. The record size and allocation of various data fields are listed in Table 7–5. The entire size of each record is 1K byte. Because this is designed for a home video library, we limit the number of records to 1,000. This can be modified, if needed, by changing the program.

This program uses many of the techniques presented in this chapter and the last. These techniques use the keyboard, disk random access files, and the graphics display mode to display a pleasing image of each video tape in the library. The display used with this program is a 16-color, 640 × 480 VGA display using graphics mode 12H. The contents of one screen of information is the display of one record. The program can also print data on the printer connected to LPT1, the parallel printer port. This program does not allow changes in the display form or in the printed output. The partial program listing (FILL and EDIT are not shown) is found in Example 7–45. The complete program listing

**TABLE 7–5** Record for a videotape cataloging system

| *Field* | *Size* | *Contents* |
|---|---|---|
| 0 | 128 | Title |
| 1 | 1 | Year |
| 2 | 2 | Running time |
| 3 | 32 | Actor 1 |
| 4 | 32 | Actor 2 |
| 5 | 32 | Actor 3 |
| 6 | 1 | Rating |
| 7 | 2 | Tape number |
| 8 | 794 | Description |

is too long to include in the text. Some things to note are the way the VID.DAT file is created, and how the VIDLIB subdirectory is created if needed.

**EXAMPLE 7–45** (pp. 262–267)

```
STACK       SEGMENT STACK
            DW    1024 DUP (?)
STACK       ENDS

DATA        SEGMENT

NUMB        DW    0                 ;tape number
RUN         DW    0                 ;running time
TITL        DB    128 DUP (0)       ;title
YEAR        DB    0                 ;year
ACT1        DB    32 DUP (0)        ;actor 1
ACT2        DB    32 DUP (0)        ;actor 2
ACT3        DB    32 DUP (0)        ;actor 3
RATE        DB    0                 ;rating
NOTE        DB    794 DUP (0)       ;notes

CHAR        DB    1024 DUP (?)      ;character lookup table

VDAT        DW    80*9,7            ;top white bar
            DW    160*9+10,3        ;2 lines of cyan
            DW    64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3
            DW    64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3
            DW    64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3
            DW    64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3
            DW    160*9,3           ;end title
            DW    3,8,77,3,3,8,77,3,3,8,77,3,3,8,77,3
            DW    3,8,77,3,3,8,77,3,3,8,77,3,3,8,77,3,3,8,77,3
            DW    160*9,3           ;end tape number
            DW    4,8,76,3,4,8,76,3,4,8,76,3,4,8,76,3
            DW    4,8,76,3,4,8,76,3,4,8,76,3,4,8,76,3,4,8,76,3
            DW    160*9,3           ;end year
            DW    5,8,75,3,5,8,75,3,5,8,75,3,5,8,75,3
            DW    5,8,75,3,5,8,75,3,5,8,75,3,5,8,75,3,5,8,75,3
            DW    160*9,3           ;end running time
            DW    1,8,79,3,1,8,79,3,1,8,79,3,1,8,79,3
            DW    1,8,79,3,1,8,79,3,1,8,79,3,1,8,79,3,1,8,79,3
            DW    160*9,3           ;end rating
            DW    32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3
```

```
                DW    32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3
                DW    160*9,3               ;end actor 1
                DW    32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3
                DW    32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3
                DW    160*9,3               ;end actor 2
                DW    32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3
                DW    32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3,32,8,48,3
                DW    160*9,3               ;end actor 3
                DW    64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3
                DW    64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,3,64,8,16,
```

```
CODE        SEGMENT  'CODE'
            ASSUME   CS:CODE,DS:DATA,SS:STACK

MAIN        PROC  FAR

            CLD                          ;select auto-increment
            MOV   AX,DATA                ;address DATA segment
            MOV   DS,AX

            MOV   AX,0012H               ;select video mode
            INT   10H

            MOV   AX,1130H               ;get 8 x 8 character set
            MOV   BH,3
            INT   10H

            MOV   CX,1024
            MOV   DI,OFFSET CHAR
            MOV   SI,BP
MAIN1:                                   ;get character table
            MOV   AL,ES:[SI]
            MOV   [DI],AL
            INC   SI
            INC   DI
            LOOP  MAIN1

            MOV   AX,0A000H              ;address video memory
            MOV   ES,AX

            CALL  PAINT                  ;paint graphics screen and labels

            MOV   AX,3D02H               ;open file
            MOV   DX,OFFSET FILEN
            INT   21H
            JNC   MAIN4
            CMP   AX,2                   ;file not found
            JE    MAIN3
            CMP   AX,3                   ;path not found
            JE    MAIN2
MAIN1A:                                  ;if disk error
            MOV   SI,OFFSET M14
            CALL  DMES
TRAP:                                    ;if disk error trap
            JMP   TRAP
MAIN2:                                   ;make subdirectory VIDLIB
            MOV   AH,39H
            MOV   DX,OFFSET DIRN
            INT   21H
            JC    MAIN1A

MAIN3:                                   ;create file
            MOV   AH,3CH
            XOR   CX,CX
            MOV   DX,OFFSET FILEN
            INT   21H
            JC    MAIN1A                 ;if error
            MOV   FILEH,AX               ;save handle
            MOV   SI,OFFSET M15          ;show program busy
            CALL  DMES
            MOV   BP,1000                ;initialize record count
            MOV   NUMB,1                 ;set record number
```

```
MAIN3A:                                 ;initialize data base
            MOV     AH,40H              ;write record
            MOV     BX,FILEH            ;get handle
            MOV     CX,1024             ;byte count
            MOV     DX,OFFSET NUMB
            INT     21H
            JC      MAIN1A              ;if error
            INC     NUMB                ;increment record number
            DEC     BP                  ;decrement count
            JNZ     MAIN3A              ;repeat 1,000 times
            MOV     SI,OFFSET M16       ;clear busy
            CALL    DMES
            MOV     AX,FILEH
MAIN4:                                  ;get first record
            MOV     FILEH,AX            ;save file handle
            MOV     BX,AX
            MOV     AX,4200H            ;front of file
            XOR     DX,DX
            XOR     CX,CX
            INT     21H
            MOV     AH,3FH              ;read first record
            MOV     CX,1024
            MOV     DX,OFFSET NUMB
            INT     21H
            JC      MAIN1A              ;if error
            CALL    FILL                ;display record
            CALL    EDIT                ;enter editor
            MOV     AH,3EH              ;close file
            MOV     BX,FILEH
            INT     21H

            MOV     AX,4C00H            ;return to DOS
            INT     21H

MAIN        ENDP

PAINT       PROC    NEAR                ;paint main screen

            MOV     DI,0                ;address video RAM
            MOV     SI,OFFSET VDAT      ;address video data

            MOV     DX,3CEH             ;address bit mask register
            MOV     AL,8
            OUT     DX,AL
            INC     DX
            MOV     AL,0FFH             ;select all PELs
            OUT     DX,AL
            MOV     DX,3C4H             ;address map mask register
            MOV     AL,2
            OUT     DX,AL
            INC     DX
PAINT1:
            LODSW
            OR      AX,AX
            JZ      PAINT3              ;if end video data
            MOV     CX,AX               ;get byte count
            LODSW                       ;get color
            MOV     BL,AL
PAINT2:
            MOV     AL,0FH
            OUT     DX,AL               ;enable all bit planes
```

```
                MOV     AL,ES:[DI]              ;load latch
                MOV     BYTE PTR ES:[DI],0      ;black
                MOV     AL,BL
                OUT     DX,AL                   ;select color
                MOV     AL,0FFH
                STOSB
                LOOP    PAINT2
                JMP     PAINT1                  ;get next
PAINT3:                                         ;display text data
                MOV     BP,OFFSET MLST          ;address text data
PAINT4:
                MOV     SI,DS:[BP]
                ADD     BP,2
                OR      SI,SI                   ;if end of text data
                JZ      PAINT5
                CALL    DMES                    ;display string
                JMP     PAINT4                  ;get next text string
PAINT5:
                RET

PAINT           ENDP

DMES            PROC    NEAR                    ;display message

                LODSB                           ;DS:SI = address of string
                MOV     CL,AL                   ;get count
                XOR     CH,CH
                LODSB                           ;get line number
                MOV     DH,AL
                LODSB                           ;get column number
                MOV     DL,AL
                LODSB                           ;get text color
                MOV     BL,AL
DMES1:
                LODSB                           ;get character
                CALL    DCHAR                   ;display character
                INC     DL                      ;address next column
                LOOP    DMES1                   ;repeat
                RET

DMES            ENDP

DCHAR           PROC    NEAR                    ;display character

                PUSH    SI                      ;save registers
                PUSH    DX
                PUSH    CX
                PUSH    AX                      ;AL = ASCII character
                MOV     CX,720                  ;BL = character color
                MOV     AL,DH                   ;DH = line number
                XOR     AH,AH                   ;DL = column number
                PUSH    DX
                MUL     CX
                POP     DX
                MOV     DI,AX
                XOR     DH,DH
                ADD     DI,DX
                MOV     SI,OFFSET CHAR          ;address character
                POP     AX
                XOR     AH,AH
                SHL     AX,1
                SHL     AX,1
```

```
                SHL     AX,1
                ADD     SI,AX
                MOV     CX,8
DCHAR1:
                MOV     DX,3CEH             ;address bit mask register
                MOV     AL,8
                OUT     DX,AL
                INC     DX
                LODSB                       ;get scan line
                OUT     DX,AL               ;set mask
                MOV     DX,3C4H             ;address map mask register
                MOV     AL,2
                OUT     DX,AL
                INC     DX
                MOV     AL,0FH              ;enable all bit planes
                OUT     DX,AL
                MOV     AL,ES:[DI]          ;load latch
                MOV     BYTE PTR ES:[DI],0  ;black
                MOV     AL,BL
                OUT     DX,AL               ;set color
                MOV     BYTE PTR ES:[DI],0FFH
                ADD     DI,80
                LOOP    DCHAR1
                POP     CX
                POP     DX
                POP     SI
                RET

DCHAR           ENDP

FILL            PROC    NEAR                ;not shown

;This procedure displays the data for the current record

FILL            ENDP

EDIT            PROC    NEAR                ;not shown

;This procedure edits the screen and responds to the function keys

EDIT            ENDP

CODE            ENDS

                END     MAIN
```

## 7–6 SUMMARY

1. The assembler program assembles modules that contain PUBLIC variables and segments plus EXTRN (external) variables. The linker program links modules and library files to create a run-time program executed from the DOS command line. The run-time program usually has the extension EXE, but may also have the extension .COM if it is short and changed to a command file with the EXE2BIN program.
2. The MACRO and ENDM directives create a new opcode for use in programs. Macros are similar to procedures except there is no call or return instruction. Instead of using

a call or return instruction, the assembler inserts the code of the macro sequence into a program each time it is invoked. Macros can include variables that pass information and data to the macro sequence.

3. The DOS INT 21H function call provides a method of using the keyboard and video display. Function number 06H, placed into register AH, provides an interface to the keyboard and display. If DL = 0FFH, this function tests the keyboard for a keystroke. If no keystroke is detected, it returns with an equal condition indicated by the zero flag. If a keystroke is detected, the standard ASCII character returns in AL. If an extended ASCII character is typed, it returns with AL = 00H, where the function must again be called to return with the extended ASCII character in AL. To display a character, DL is loaded with the character and AH with 06H before the INT 21H is used in a program.
4. Character strings are displayed using function number 09H. The DS:DX register combination addresses the character string, which must end with a $ or 24H.
5. The INT 10H instruction accesses BIOS (basic I/O system) procedures that control the video display and keyboard. The BIOS functions are independent of DOS and function with any operating system.
6. Data conversion from binary to BCD is accomplished with the AAM instruction for numbers that are less than 100, or by repeated division by 10 for larger numbers. Once converted to BCD, a 30H is added to convert each digit to ASCII code for the video display.
7. When converting from an ASCII number to BCD, a 30H is subtracted from each digit. To obtain the binary equivalent, multiply by 10.
8. Lookup tables are used for code conversion, with the XLAT instruction if the code is an 8-bit code. If the code is wider than eight bits, then a short procedure that accesses a lookup table provides the conversion. Lookup tables are also used to hold addresses so that different parts of a program or different procedures can be selected.
9. The video display is accessed through the video BIOS or directly through the video display memory. For a VGA display adapter, the video display memory exists at locations A0000H–AFFFFH for the graphics modes 12H and 13H. Mode 12H is a 16-color, 640 × 480 display mode. Mode 13H is a 256-color, 320 × 200 display mode.
10. The 256-color mode stores one picture element per byte of memory and the 16-color mode is a bit-mapped mode storing eight PELs per memory byte in four bit planes.
11. The bit mask register selects the PEL to be written in the 16-color, 640 × 480 video display. The map mask register selects the bit planes (color) to be written in the 16-color, 640 × 480 video display.
12. The video BIOS ROM contains character sets that are copied to an application program to display text on the graphics mode VGA screen. This allows the display of standard characters, custom characters or character sets, or even multiple character sets.
13. Two types of floppy disks are found in application today: the mini-floppy 5¼″ disk, which stores 360K bytes (DSDD) or 1.2M bytes (HD), and the micro-floppy 3½″ disk, which stores 720K bytes (DSDD), 1.44M bytes (HD), or 2.88M bytes (EHD or ED).
14. Disk data are stored in sectors that often contain 512 bytes of data. A track is a concentric ring that holds nine or more sectors of data. A cylinder contains two or more tracks. A cluster often contains four or more sectors.
15. Data are stored on a disk starting with the boot sector located in the outermost track at its first sector. Following the boot sector, which contains the bootstrap loader pro-

gram, is the FAT (file allocation table) that shows which sectors (clusters) are occupied and which are free. Located after the FAT is the root directory, which contains the system programs entries and root directory. The remaining part of the disk contains files and subdirectories.

16. Files are accessed by first opening or creating them using DOS function 21H calls. Once a file is opened or created, it can be read or written using other DOS function 21H calls. Before the program that uses a file ends, the file must be closed or problems can result. Files are accessed by the use of a file name stored in an ASCII-Z string. The ASCII-Z string is an ASCII character string that ends with a NUL (00H).
17. Once a file is opened via an ASCII-Z string, it is referred to by a file handle. The file handle allows DOS to process file references more efficiently.
18. The file pointer is a 32-bit number used to refer to any byte in a file. A DOS function 21H call, which allows the file pointer to be moved any number of bytes from the start of the file, the current location, or the end of the file, is provided.
19. Random access files are created by using sequential access files and the file pointer. The file pointer can be used to randomly access any portion of the sequential file.

## 7–7 QUESTIONS AND PROBLEMS

1. The assembler converts a source file into an _____ file.
2. What files are generated from the source file TEST.ASM as it is processed by MASM?
3. The linker program links object files and _____ files to create an execution file.
4. What is the difference between an .EXE and a .COM file?
5. What does the PUBLIC directive indicate when placed in a program module?
6. What does the EXTRN directive indicate when placed in a program module?
7. What directive appears when a label external to the current programming module is defined?
8. Describe how a library file works when it is linked to other object files by the linker program.
9. What assembler language directives delineate a macro sequence?
10. What is a macro sequence?
11. How are parameters transferred to a macro sequence?
12. Develop a macro called ADD32 that adds the 32-bit contents of DX–CX to the 32-bit contents of BX–AX.
13. Develop a macro called STUB that sign extends the 8-bit number in AL into a 64-bit sign-extended number in ECX–EBX. (Note that ECX is the most significant part of the 64-bit result.)
14. How is the LOCAL directive used within a macro sequence?
15. Develop a macro called ADDLIST PARA1,PARA2 that adds the contents of PARA1 to PARA2. Each of these parameters represents an area of memory. The number of bytes added are indicated by register CX before the macro is invoked.

16. Develop a macro that sums a list of byte-sized data invoked by the macro ADDM LIST,LENGTH. The label LIST is the starting address of the data block and length is the number of data added. The result must be a 16-bit sum found in AX at the end of the macro sequence.
17. What is the purpose of the INCLUDE directive?
18. Develop a procedure called RANDOM. This procedure must return an 8-bit random number in register CL at the end of the subroutine. (One way to generate a random number is to increment CL each time the DOS function 06H tests the keyboard and finds *no* keystroke.)
19. Modify the procedure developed in Question 18 so the random number ranges in value from 1 through 6.
20. Develop a procedure that displays a character string that ends with a 00H. Your procedure must use the DS:DX register to address the start of the character string.
21. Develop a procedure that reads a key and displays the hexadecimal value of an extended ASCII-coded keyboard character if it is typed. If a normal character is typed, ignore it.
22. Use BIOS INT 10H to develop a procedure that positions the cursor at line 3, column 6.
23. When a number is converted from binary to BCD, the _____ instruction accomplishes the conversion provided the number is less than 100 decimal.
24. How is a large number (over 100 decimal) converted from binary to BCD?
25. A BCD digit is converted to ASCII code by adding a _____.
26. An ASCII-coded number is converted to BCD by subtracting _____.
27. Develop a procedure that reads an ASCII number from the keyboard and stores it as a BCD number into memory array DATA. The number ends when anything other than a number is typed.
28. Explain how a 3-digit ASCII-coded number is converted to binary.
29. Develop a procedure that converts all lowercase ASCII-coded letters into uppercase ASCII-coded letters. This procedure may not change any other character except the letters a–z.
30. Develop a lookup table that converts hexadecimal data 00H–0FH into the ASCII-coded characters that represent the hexadecimal digits. Make sure to show the lookup table and any software required for the conversion.
31. Develop a program sequence that jumps to memory location ONE if AL = 6, TWO if AL = 7, and THREE if AL = 8.
32. Show how to use the XLAT instruction to access a lookup table called LOOK, located in the stack segment.
33. Write a program that reads any decimal number between 0 and 65,535 and displays the 16-bit binary version on the video display.
34. Write a program that displays the binary powers of two (in decimal) on the video screen for the powers 0 through 7. Your display must show $2^n$ = value for each power of 2.
35. Using the technique learned in Question 18, develop a program that displays random numbers between 1 and 47 (or whatever) for your state's lottery.
36. Develop a program that displays the hexadecimal contents of a block of 256 bytes of memory. Your software must be able to accept the starting address as a hexadecimal number between 00000H and FFF00H.

37. Where is the video data stored when mode 13H is selected for the VGA video display adapter?
38. How many bytes of memory are required to store a screen's worth of information for the 256-color, 320 × 200 video display mode?
39. What is a bit plane?
40. How is the bit mask register used in the 16-color, 640 × 480 VGA video display mode?
41. How is the map mask register used in the 16-color, 640 × 480 VGA video display mode?
42. Develop a program that places the video adapter into mode 13H and then draws a green line, 2 PELs wide, from the upper left-most corner to the lower right-most corner.
43. Develop a program that places the video adapter into mode 13H and then displays a red letter T on the screen, 50 PELs high and 30 PELs wide. The width of the lines in the letter T should be 3 PELs wide.
44. Develop a program that places the video adapter into mode 12H and then displays a vertical color bar pattern that displays 16 color bars (each 40 PELs wide) showing the 16 colors available in this mode.
45. Develop a program that places the video adapter into mode 12H and then displays a white screen (color 7) with a cyan horizontal bar 10 scanning lines high across the top.
46. Develop your own character set for mode 12H of the VGA display. Your character set must contain characters that are 16 bits wide and 16 bits high. Luckily you only need the characters N, a, m, e, space, and =.
47. Using the characters developed in Question 46, develop a program using mode 12H that displays a blue video screen that contains the red character line "Name = " at the upper left margin of the screen.
48. A 5¼″ floppy disk is called a _____ floppy disk.
49. A 3½″ floppy disk is called a _____ floppy disk.
50. How much data is stored on a 5¼″ double-sided, double-density disk?
51. The high density 3½″ floppy disk stores _____ bytes of data.
52. How much information is stored on a 3½″ extra high density disk?
53. Describe the following terms:
    (a) Track
    (b) Sector
    (c) Cluster
    (d) Cylinder
    (e) FAT
54. What information is stored i n the file allocation table?
55. Detail the purpose of the boot sector.
56. The root directory contains _____ entries on a floppy disk.
57. How many directory entries can be placed in a subdirectory?
58. How many characters can be used for a file name?
59. How many characters can be used in a file extension?
60. How many bytes are used to store a directory name?
61. Develop a short sequence of instructions that opens a file, whose name is stored at data-segment offset address FILEN, as a read-only file.
62. Develop a short sequence of instructions that creates a hidden file, whose name is stored at data-segment offset address FILES.

63. Develop a procedure that creates a file whose name is FROG.LST and fills it with 512 bytes of 00H. (Don't forget to close the file.)
64. Write a macro instruction sequence called READ. The syntax for this macro is READ BUFFER,COUNT,HANDLE where BUFFER is the offset address of a file buffer, COUNT is the number of bytes read, and HANDLE is the address of the file handle.
65. Develop a macro sequence called APPEND. The syntax for this macro is APPEND BUFFER,COUNT,HANDLE where BUFFER is the offset address of the file buffer, COUNT is the number of bytes to append to the file, and HANDLE is the address of the file handle. (Don't forget to use the move file pointer function to append the end of the file.)
66. Create a random access file (RAN.FIL) containing 200 records with a record length of 100 bytes. To initialize this random access file, place a 00H in every byte.
67. Using the random access file developed in Question 66, develop a macro sequence that accesses and reads any record. The syntax for this macro is ACCESS BUFFER, RECORD,HANDLE where BUFFER is the offset address of the record buffer, RECORD is the record number, and HANDLE is the address of the file handle.
68. Modify the macro of Question 67 so it can read or write a record. The syntax is ACCESS T,BUFFER,RECORD,HANDLE where T is added to specify a read 'R' or a write 'W' using the ASCII characters for the letters 'R' and 'W'.
69. Develop a short program that displays the first 256 bytes of a file on the video screen. Your display must appear as in Example 7–46, showing the actual data in any file. It must ask for the name of the file to be displayed, as in the example, and then display the first 256 bytes. (Use text mode.)

**EXAMPLE 7–46**

```
FILENAME = TEST.TXT
00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

70. Redo Question 69 using graphics mode to display black numerals on a cyan video screen.

# CHAPTER 8

# The Intel Family of Arithmetic Coprocessors

## INTRODUCTION

The Intel family of arithmetic coprocessors includes the 8087, 80287, 80387SX, 80387DX, and the 80487SX for use with the 80486SX microprocessor. The 80486DX and Pentium microprocessors both contain their own built-in arithmetic coprocessors. The instruction sets and programming for all devices are almost identical, but each coprocessor is designed to function with a different Intel microprocessor. This chapter provides detail on the entire family of arithmetic coprocessors. Because the coprocessor is a part of the 80486DX and Pentium, and because these microprocessors are commonplace, many programs now require, or at least benefit from, a coprocessor.

The family of coprocessors that we will label the 80X87 is able to multiply, divide, add, subtract, and find the square root, partial tangent, partial arctangent, and logarithms. Data types include: 16-, 32-, and 64-bit signed integers; 18-digit BCD data; and 32-, 64-, and 80-bit floating-point numbers. The operations performed by the 80X87 generally execute about 100 times faster than equivalent operations using the microprocessor's normal instruction set. With the improved Pentium coprocessor, operations execute about 500 times faster than those performed by equivalent software. Note that the Pentium can often execute a coprocessor instruction and two integer instructions simultaneously.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Convert between decimal data and signed-integer, BCD, and floating-point data for use by the arithmetic coprocessor.
2. Explain the operation of the 80X87 arithmetic coprocessor.
3. Explain the operation and addressing modes of each arithmetic coprocessor instruction.
4. Develop programs that solve complex arithmetic problems using the arithmetic coprocessor.

# 8–1 DATA FORMATS FOR THE ARITHMETIC COPROCESSOR

This section of the text presents the types of data used with all coprocessor family members. (Refer to Table 8–1 for a listing of all Intel microprocessors and their companion coprocessors.) These data types include signed-integer, BCD, and floating-point. Each has a specific use in a system, and many systems require all three data types.

## Signed Integers

The signed integers used with the coprocessor are basically the same as those described in Chapter 1. When used with the arithmetic coprocessor, signed integers are 16 (*word*) , 32 (*short integer*), or 64 bits (*long integer*) in width. Conversion between decimal and signed-integer format is handled exactly the same as for the signed integers described in Chapter 1. As you will recall, positive numbers are stored in true form with a left-most sign bit of 0, and negative numbers are stored in two's complement form with a left-most sign bit of 1.

The word integers range in value from –32,768 to +32,767, the short integer range is $\pm 2 \times 10^{+9}$, and the long integer range is $\pm 9 \times 10^{+18}$. Integer data types are found in some applications that use the arithmetic coprocessor. Refer to Figure 8–1, which shows these three forms of signed-integer data.

**TABLE 8–1** Microprocessor and Intel coprocessor compatibility

| *Microprocessor* | *Coprocessor* |
|---|---|
| 8086 | 8087 |
| 8088 | 8087 |
| 80286 | 80287 |
| 80386SX | 80387SX |
| 80386DX | 80387DX |
| 80486SX | 80487 |
| 80486DX | Built into microprocessor |
| Pentium | Built into microprocessor |

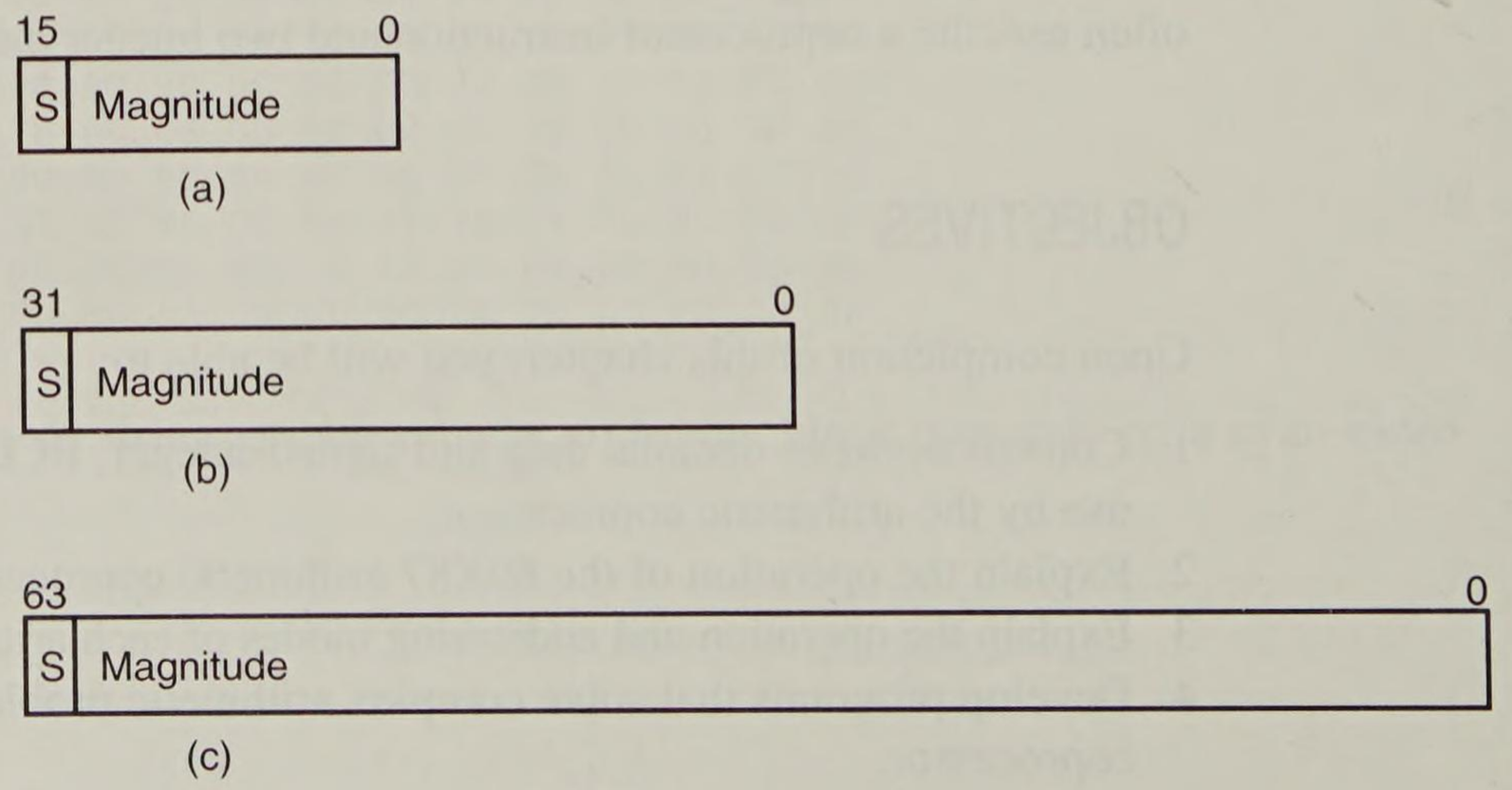



**FIGURE 8–1** Integer forms of data for the 8087 family of numeric coprocessors. (a) Word, (b) short, and (c) long.

Data are stored in memory using the same assembler directives described and used in earlier chapters. We use the DW directive to define words, DD to define short integers, and DQ to define long integers. Example 8–1 shows how several different sizes of signed integers are defined for use by the assembler and arithmetic coprocessor.

**EXAMPLE 8–1**

```
0000 0002              DATA1  DW  +2              ;16-bit integer
0002 FFDE              DATA2  DW  -34             ;16-bit integer
0004 000004D2          DATA3  DD  +1234           ;short integer
0008 FFFFFF9C          DATA4  DD  -100            ;short integer
000C 0000000000005BA0  DATA5  DQ  +23456          ;long integer
0014 FFFFFFFFFFFFFF86  DATA6  DQ  -122            ;long integer
```

## Binary-Coded Decimal (BCD)

The binary-coded decimal (BCD) form requires 80 bits of memory. Each number is stored as an 18-digit packed integer in 9 bytes of memory as two digits per byte. The tenth byte contains only a sign bit for the 18-digit signed BCD number. Figure 8–2 shows the format of the BCD number used with the arithmetic coprocessor. Note that both positive and negative numbers are stored in true form, never in 10's complement form. The DT directive stores BCD data in the memory as illustrated in Example 8–2.

**EXAMPLE 8–2**

```
0000                   DATA1  DT   200            ;200 decimal stored as BCD
     0000000000000000200

000A                   DATA2  DT   -10            ;-10 decimal stored as BCD
     8000000000000000010

0014                   DATA3  DT   10020          ;10,020 decimal stored as BCD
     0000000000000010020
```

## Floating-Point Numbers

Floating-point numbers are often called *real numbers* because they hold signed integers, fractions, and mixed numbers. A floating-point number has three parts: a *sign bit*, a *biased exponent*, and a *significand*. Floating-point numbers are written in *scientific binary notation*. The Intel family of arithmetic coprocessors supports three types of floating-point numbers: short (32 bits), long (64 bits), and temporary (80 bits). Refer to Figure 8–3 for the three forms of the floating-point number. Please note that we also call the short form a *single-precision* number and the long form a *double-precision number*. Sometimes the 80-bit temporary form is called an *extended-precision* number. The floating-point numbers, and the operations performed by the arithmetic coprocessor, conform to the IEEE-754 standard

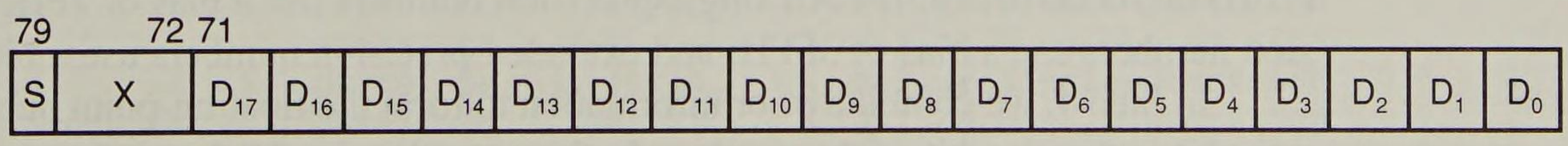



**FIGURE 8–2** BCD data format for the 8087 family of numeric coprocessors.

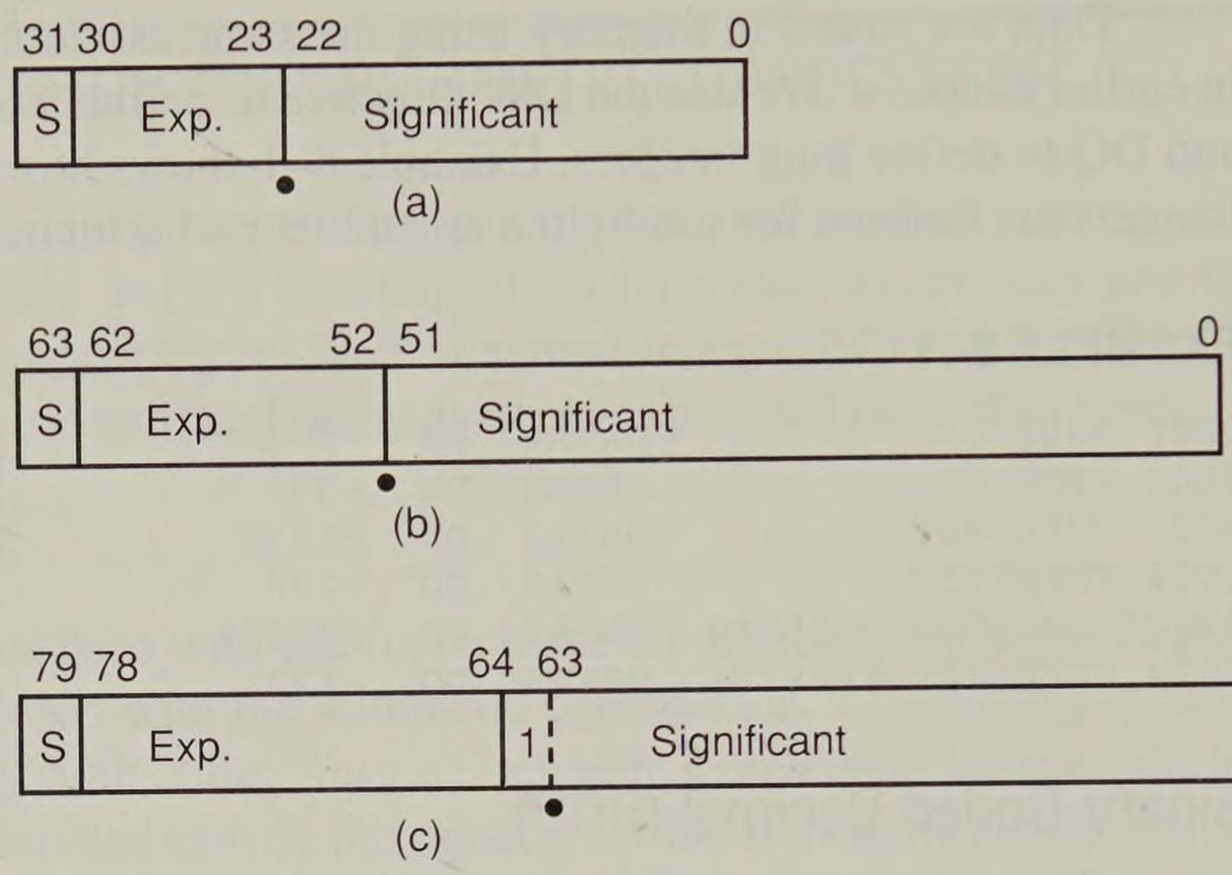



**FIGURE 8–3** Floating-point (real) numbers. (a) Short with a bias of 127 (7FH) and an implicit 1, (b) long with a bias of 1023 (3FFH) and an implicit 1, and (c) temporary with a bias of 16,383 (3FFFH) and a non-implicit 1.

adopted by all major software producers. This includes Microsoft, which has recently stopped supporting the Microsoft floating-point format and the ANSI floating point standard popular in mainframe computer systems.

*Converting to Floating-Point Form.* Converting from decimal to floating-point form is a simple task that is accomplished through the following steps:

1. Convert the decimal number into binary.
2. Normalize the binary number.
3. Calculate the biased exponent.
4. Store the number in the floating-point format.

These four steps are illustrated for the decimal number 100.25 in Example 8–3. Here the decimal number is converted to a single-precision, 32-bit, floating-point number.

**EXAMPLE 8–3**

| Step | Result |
|---|---|
| 1 | 100.25 = 1100100.01 |
| 2 | 1100100.01 = $1.10010001 \times 2^6$ |
| 3 | 110 + 01111111 = 10000101 |
| 4 | Sign = 0<br>Exponent = 10000101<br>Significand = 10010001000000000000000 |

In Step 3 the biased exponent is the exponent, a +6 (110), plus a bias of 01111111 (7FH) or 10000101 (85H). All single-precision numbers use a bias of 7FH; double-precision numbers use a bias of 3FFH; and extended-precision numbers use a bias of 3FFFH.

In Step 4, we combine prior information to form the floating-point number. The leftmost bit is the sign bit of the number. In this case the sign bit is a 0, because the number is +100.25. The biased exponent follows the sign bit. The significand is a 23-bit number

with an implied one-bit. Note that the significand of a number, 1.XXXX, is the XXXX portion. The 1. is an implied one-bit that is stored in the extended-precision form of the floating-point number as an explicit one-bit.

Some special rules apply to a few numbers. The number 0, for example, is stored as all zeros except for the sign bit, which can be a logic 1 to represent a negative zero. We also store plus and minus infinity as logic 1s in the exponent with a significand of all zeros and a sign bit to represent plus or minus. A NAN (*not-a-number*) is an invalid floating-point result that has all ones in the exponent with a significand that is *not* all zeros.

*Converting from Floating-Point Form.* Conversion to a decimal number from a floating-point number is summarized in the following steps:

1. Separate the sign bit, biased exponent, and significand.
2. Convert the biased exponent into a true exponent by subtracting the bias.
3. Write the number as a normalized binary number.
4. Convert it to a denormalized binary number.
5. Convert the denormalized binary number to decimal.

These five steps convert a single-precision floating-point number to decimal in Example 8–4. Notice how the sign bit of 1 makes the decimal result negative. Also notice that the implied one-bit is added to the normalized binary result in Step 3.

**EXAMPLE 8–4**

```
Step        Result

1           Sign = 1
            Exponent = 10000011
            Significand = 10010010000000000000000

2           100 = 10000011 - 01111111

3           1.1001001 × 2^4

4           11001.001

5           -25.125
```

*Storing Floating-Point Data in Memory.* Floating-point data are stored with the assembler using the DD directive for single-precision numbers, DQ for double-precision numbers, and DT for extended-precision numbers. Some examples of floating-point data storage are shown in Example 8–5. The author discovered that the Microsoft version 6.0 macro assembler contains an error that does not allow a plus sign to be used with positive floating-point numbers. A +92.45 must be defined as 92.45 for the assembler to function correctly. Microsoft has assured the author that this error has been corrected in version 6.1 of MASM.

**EXAMPLE 8–5**

```
0000 C377999A                DATA7  DD   -247.6       ;define single-precision
0004 40000000                DATA8  DD   2.0          ;define single-precision
0008 486F4200                DATA9  DD   2.45E+5      ;define single-precision
000C 4059100000000000        DATA10 DQ   100.25       ;define double-precision
0014 3F543BF727136A40        DATA11 DQ   0.001235     ;define double-precision
001C 400487F34D6A161E4F76 DATA12 DT   33.9876      ;define extended-precision
```

## 8–2 THE 80X87 ARCHITECTURE

The 80X87 coprocessor is designed to operate concurrently with the microprocessor. Note that the 80486DX and Pentium microprocessors contain their own *internal* and fully compatible versions of the 80387. With other family members, the coprocessor is an external integrated circuit that parallels most of the connections of the microprocessor. The 80X87 executes 68 different instructions. The microprocessor executes all normal instructions and the 80X87 executes arithmetic coprocessor instructions. Both the microprocessor and coprocessor can execute their respective instructions simultaneously or concurrently. The numeric or arithmetic *coprocessor* is a special-purpose microprocessor that is specially designed to efficiently execute arithmetic and transcendental operations.

To illustrate one of the coprocessors, Figure 8–4 shows the pinout of the 80287 arithmetic coprocessor, sometimes called a *numeric coprocessor*. Recall that the 80287 coprocessor is a companion to the 80286 microprocessor.

The coprocessor pin connections are designed to be connected to the 80286 microprocessor in parallel. This connection allows both the microprocessor and the coprocessor to sample the data bus for instructions that apply to each device. The microprocessor intercepts and executes the normal instruction set and the coprocessor intercepts and executes only the coprocessor instructions. Recall that the coprocessor instructions are actually escape (ESC) instructions. These instructions are used by the microprocessor to generate a memory address for the coprocessor so the coprocessor can execute a coprocessor instruction.

### Internal Structure of the 80X87

Figure 8–5 shows the internal structure of the arithmetic coprocessor. Notice that this device is divided into two major sections: the *control unit* and the *numeric execution unit*.

The control unit interfaces the coprocessor to the microprocessor system data bus. Both the devices monitor the instruction stream. If the instruction is an ESCape (*coprocessor*) instruction, the coprocessor executes it, and if not, the microprocessor executes it.

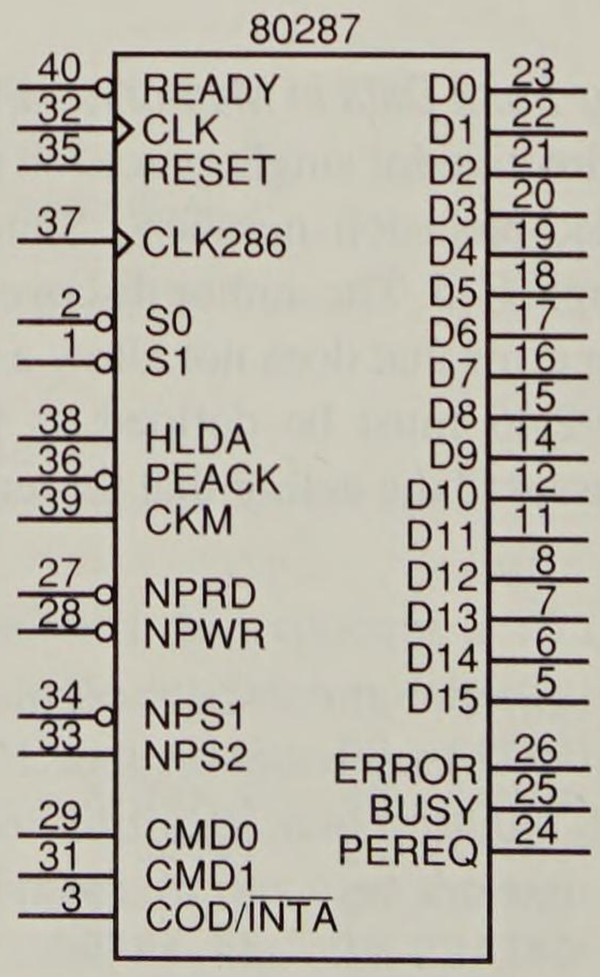



**FIGURE 8–4** The 80827 arithmetic coprocessor.

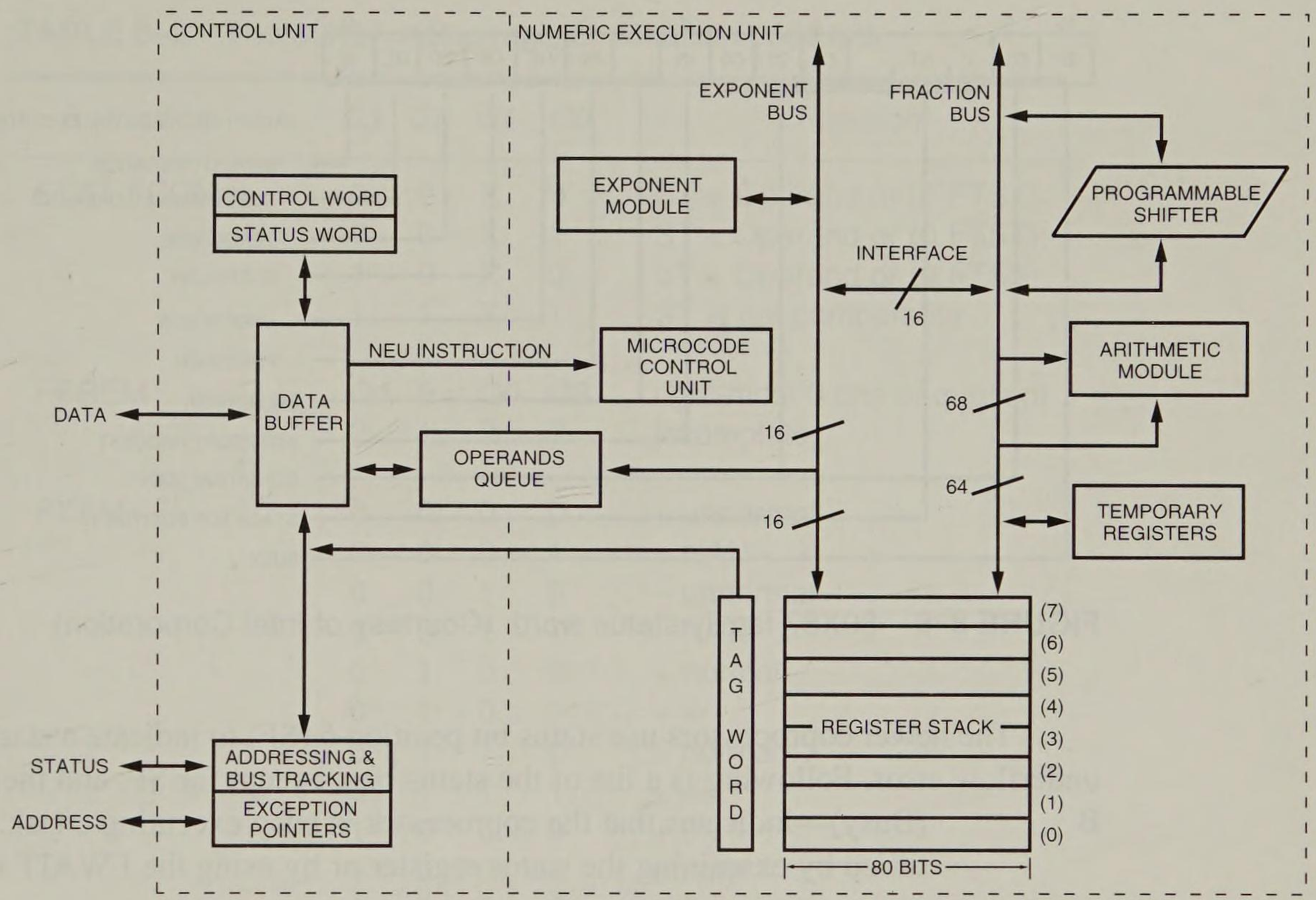



**FIGURE 8–5** The internal structure of the 80X87. (Courtesy of Intel Corporation)

The numeric execution unit (NEU) is responsible for executing all coprocessor instructions. The NEU has an eight-register stack that holds operands for arithmetic instructions and the results of arithmetic instructions. Instructions either address data in specific stack data registers or use a push and pop mechanism to store and retrieve data on the top of the stack. Other registers in the NEU are status, control, tag, and exception pointers. A few instructions transfer data between the coprocessor and the AX register.

The stack within the coprocessor contains eight registers that are each 80 bits wide. These stack registers *always* contain an 80-bit, extended-precision, floating-point number. The only time data appear as any other form is when the data reside in the memory system. The coprocessor converts from signed-integer, BCD, single-precision, or double-precision form as the data are moved between the memory and the coprocessor register stack.

*Status Register.* The status register (see Figure 8–6) reflects the overall operation of the coprocessor. The status register is accessed by executing the instruction FSTSW, which stores the contents of the status register into a word of memory. The FSTSW AX instruction copies the status register directly to the AX register on the 80287, or more recent coprocessors. Once status is stored in memory or the AX register, the bit positions of the status register can be examined by normal software. The coprocessor/microprocessor communications are carried out through the I/O ports 00FAH–00FFH on the 80287 and 32-bit I/O port 80000000H on the 80386, 80486, and Pentium.

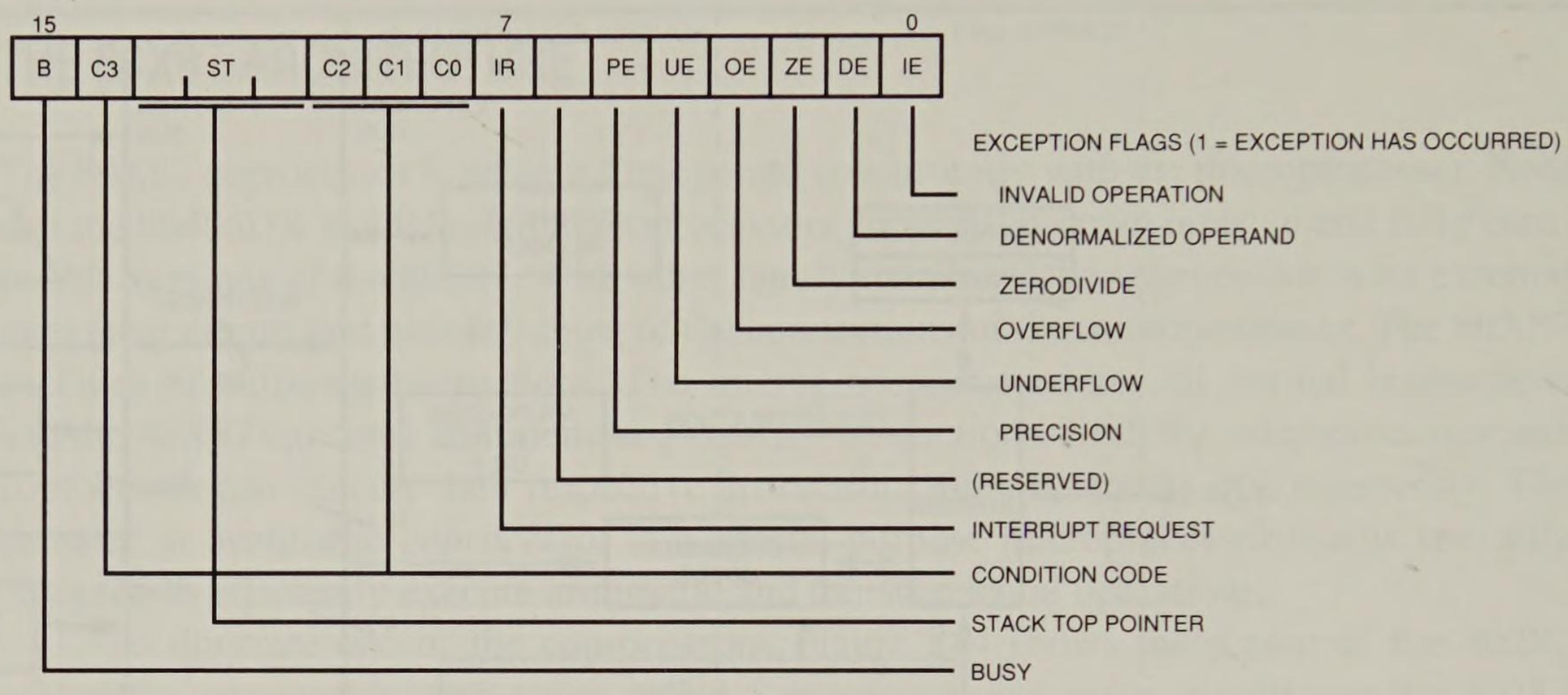



**FIGURE 8–6** 80X87 family status word. (Courtesy of Intel Corporation)

The newer coprocessors use status bit position 6 (SF) to indicate a stack overflow or underflow error. Following is a list of the status bits, except for SF, and their application:

B (Busy)—indicates that the coprocessor is busy executing a task. Busy can be tested by examining the status register or by using the FWAIT instruction.

C3–C0 (Condition Code Bits)—these bits have different meaning for different instructions as indicated in the table. Refer to Table 8–2 for a complete listing of each combination of these bits and their function.

TOP (Top-of-Stack)—indicates the current register addressed as the top of the stack.

ES (Error Summary)—is set if any unmasked error bit (PE, UE, OE, ZE, DE, or IE) is set.

PE (Precision Error)—indicates result or operands exceed selected precision.

UE (Underflow Error)—indicates a nonzero result that is too small to represent with the current precision selected by the control word.

OE (Overflow Error)—indicates a result that is too large to be represented. If this error is masked, the coprocessor generates infinity for an overflow error.

ZE (Zero Error)—indicates the divisor was zero while the dividend was a noninfinity or nonzero number.

DE (Denormalized Error)—indicates at least one of the operands is denormalized.

IE (Invalid Error)—indicates a stack overflow or underflow, indeterminate form (0/0, ∞, – ∞, etc.), or the use of a NAN as an operand. This flag indicates errors such as those produced by taking the square root of a negative number, etc.

**EXAMPLE 8–6**

```
                                ;using TEST to isolate the divide by zero error bit

0000 67& D8 35 00000000 R       FDIV        DATA1
0007 9B DF E                    FSTSW       AX
000A A9 000                     TEST        AX,4
```

**TABLE 8–2** The 80X87 status register condition code bits

| *Instruction* | *C3* | *C2* | *C1* | *C0* | *Function* |
|---|---|---|---|---|---|
| FTST,FCOM | 0 | 0 | X | 0 | ST > Operand or (0 FTST) |
| | 0 | 0 | X | 1 | ST < Operand or (0 FTST) |
| | 1 | 0 | X | 0 | ST = Operand or (0 FTST) |
| | 1 | 1 | X | 1 | ST is not comparable |
| FPREM | Q1 | 0 | Q0 | Q2 | right-most 3 bits of quotient |
| | ? | 1 | ? | ? | incomplete |
| FXAM | 0 | 0 | 0 | 0 | + unnormal |
| | 0 | 0 | 0 | 1 | + NAN |
| | 0 | 0 | 1 | 0 | – unnormal |
| | 0 | 0 | 1 | 1 | – NAN |
| | 0 | 1 | 0 | 0 | + normal |
| | 0 | 1 | 0 | 1 | + ∞ |
| | 0 | 1 | 1 | 0 | – normal |
| | 0 | 1 | 1 | 1 | – ∞ |
| | 1 | 0 | 0 | 0 | + 0 |
| | 1 | 0 | 0 | 1 | empty |
| | 1 | 0 | 1 | 0 | – 0 |
| | 1 | 0 | 1 | 1 | empty |
| | 1 | 1 | 0 | 0 | + denormal |
| | 1 | 1 | 0 | 1 | empty |
| | 1 | 1 | 1 | 0 | – denormal |
| | 1 | 1 | 1 | 1 | empty |

*Notes*: Unnormal = leading bits of the significand are zero
Denormal = exponent at its most negative value
Normal = standard floating-point form
NAN (not-a-number) = an exponent of all ones and a significand not equal to zero

```
000D 75 18                      JNZ         DIVIDE_ERROR

                        ;using TEST to isolate the invalid operation error bit
                        ;after a FSQRT instruction

000F D9 FA                      FSQRT
0011 9B DF E0                   FSTSWAX
0014 A9 0001                    TEST        AX,1
0017 75 0E                      JNZ         FSQRT_ERROR

                        ;using the SAHF instruction and conditional jumps to
                        ;test for the conditions in Table 8-3 after an FCOM

0019 67& D8 15 00000000 R       FCOM        DATA1
0020 9E                         SAHF
0021 74 04                      JE          ST_EQUAL
0023 72 02                      JB          ST_BELOW
0025 77 00                      JA          ST_ABOVE
```

There are two ways to test the bits of the status register once they are moved into the AX register. One method uses the TEST instruction to test individuals bits of the status register. The other uses the SAHF instruction to transfer the left-most 8 bits of the status register into the microprocessor's flag register. Both methods are illustrated in Example 8–6. Note that the conditional jump instructions are used with the SAHF instruction to test for the condition listed in Table 8–3. Although SAHF and conditional jumps cannot test all possible operating conditions of the coprocessor, they can help to reduce the complexity of certain tested conditions. Note that SAHF places C0 into the carry flag, C2 into the parity flag, and C3 into the zero flag.

When the FXAM instruction and FSTSW AX are executed and followed by the SAHF instruction, the zero flag will contain C3. Notice from Table 8–2 that C3 indicates a ±0 when set along with other error. The FXAM instruction could be used to test a divisor for a zero value before division, by using the JZ instruction following FXAM, FSTSW AX, and SAHF.

*Control Register.* The control register is pictured in Figure 8–7. The control register selects precision, rounding control, and infinity control. It also masks and unmasks the exception bits that correspond to the right-most 6 bits of the status register. The FLDCW instruction is used to load a value into the control register.

Following is a description of each bit or grouping of bits found in the control register:

IC (Infinity Control)—selects either affine or projective infinity. Affine allows positive and negative infinity, while projective assumes infinity is unsigned.

RC (Rounding Control)—selects the type of rounding as defined in Figure 8–7.

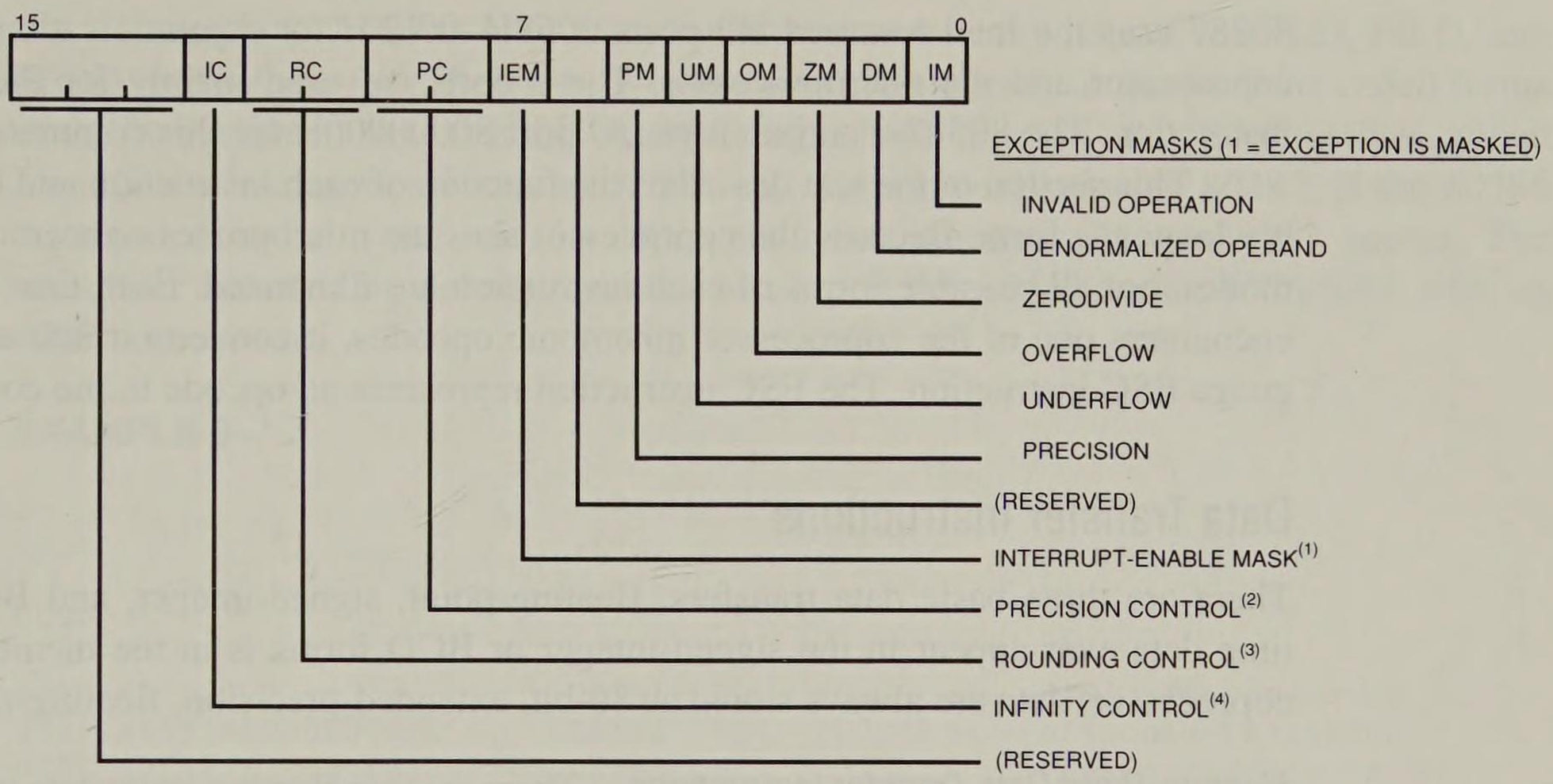
PC (Precision Control)—selects the precision of the result as defined in Figure 8–7.

EM (Exception Masks)—determine whether the error indicated by the exception affects the error bit in the status register. If a logic 1 is placed in one of the exception control bits, the corresponding status register bit is masked off.

*Tag Register.* The tag register indicates the contents of each location in the coprocessor stack. Figure 8–8 illustrates the tag register and the status indicated by each tag. The tag indicates whether a register is: valid; zero; invalid or infinity; or empty. The only way a program can view the tag register is by storing the coprocessor environment using the FSTENV, FSAVE, or FRSTOR instructions. Each of these instructions stores the tag register along with other coprocessor data.

**TABLE 8–3** Coprocessor conditions tested with the conditional jump instructions and SAHF after FCOM or FTST, as illustrated in Example 8–6

| *C3* | *C2* | *C0* | *Condition* | *Jump Instruction* |
|---|---|---|---|---|
| 0 | 0 | 0 | ST > Operand | JA (jump if ST above) |
| 0 | 0 | 1 | ST < Operand | JB (jump if ST below) |
| 1 | 0 | 0 | ST = Operand | JE (jump if ST equal) |




(1) Interrupt-Enable Mask:
0 = Interrupts Enabled
1 = Interrupts Disabled (Masked)

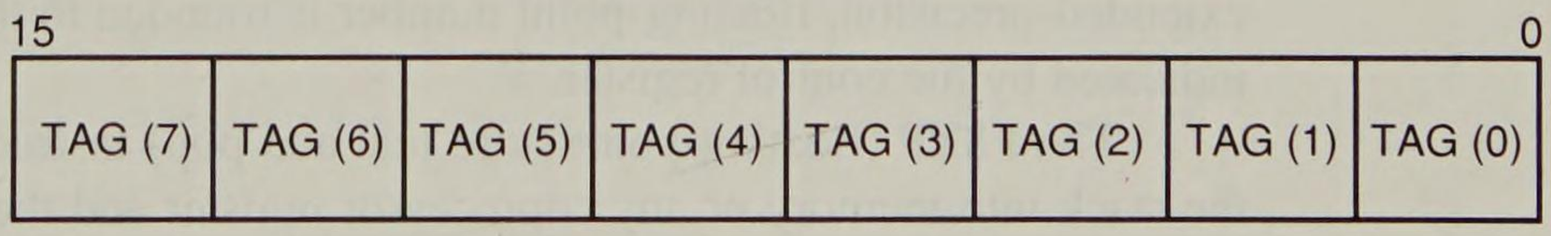
(2) Precision Control:
00 = 24 bits
01 = (reserved)
10 = 53 bits
11 = 64 bits

(3) Rounding Control:
00 = Round to Nearest or Even
01 = Round Down (toward $-\infty$)
10 = Round Up (toward $+\infty$)
11 = Chop (Truncate toward Zero)

(4) Infinity Control:
0 = Projective
1 = Affine

**FIGURE 8–7** 80X87 family control register. (Courtesy of Intel Corporation)

**FIGURE 8–8** The TAG register and conditions held in each TAG. (Courtesy of Intel Corporation)




NOTE:
The index i of tag(i) is not top-relative. A program typically uses the "top" field of Status Word to determine which tag(i) field refers to logical top of stack.

TAG VALUES:
00 = VALID
01 = ZERO
10 = INVALID or INFINITY
11 = EMPTY

## 8–3 INSTRUCTION SET

The arithmetic coprocessor executes 68 different instructions. Whenever a coprocessor instruction references memory, the microprocessor automatically generates the memory address for the instruction. The coprocessor uses the data bus for data transfers during coprocessor instructions and the microprocessor uses it during normal instructions. The

80287 uses the Intel reserved I/O ports 00F8H–00FFH for communications between the coprocessor and the microprocessor. These ports are used mainly for the FSTSW AX instruction. The 80387–Pentium uses I/O port 80000000H for this communication.

This section of the text describes the function of each instruction and lists its assembly language form. Because the coprocessor uses the microprocessor memory addressing modes, not all possible forms of each instruction are illustrated. Each time the assembler encounters one of the coprocessor mnemonic opcodes, it converts it into a machine language ESC instruction. The ESC instruction represents an opcode to the coprocessor.

## Data Transfer Instructions

There are three basic data transfers: floating-point, signed-integer, and BCD. The only time data ever appear in the signed-integer or BCD forms is in the memory. Inside the coprocessor, data are always stored as 80-bit, extended-precision, floating-point numbers.

*Floating-Point Data Transfer Instructions.* There are four floating-point data transfer instructions in the coprocessor instruction set: FLD (load real), FST (store real), FSTP (store real and pop), and FXCH (exchange).

The FLD instruction loads floating-point memory data to the top of the internal stack. This instruction stores the data on the top of the stack and then decrements the stack pointer by one. Data loaded to the top of the stack are from any memory location or from another coprocessor register. For example, an FLD ST(2) instruction copies the contents of register 2 to the stack top, which is ST. The top of the stack is register 0 when the coprocessor is reset or initialized. Another example is the FLD DATA7 instruction, which copies the contents of memory location DATA7 to the top of the stack. The size of the transfer is automatically determined by the assembler through the directives DD for single-precision numbers, DQ for double-precision numbers, and DT for extended-precision numbers.

The FST instruction stores a copy of the top of the stack into the memory location or coprocessor register indicated by the operand. At the time of storage, the internal, extended-precision, floating-point number is rounded to the size of floating-point number indicated by the control register.

The FSTP (floating-point store real and pop) instruction stores a copy of the top of the stack into memory or any coprocessor register and then pops the data from the top of the stack. You might think of FST as a *copy* instruction and FSTP as a *removal* instruction.

The FXCH instruction exchanges the register indicated by the operand with the top of the stack. For example, the FXCH ST(2) instruction exchanges the top of the stack with register 2.

*Integer Data Transfer Instructions.* The coprocessor supports three integer data transfer instructions: FILD (load integer), FIST (store integer), and FISTP (store integer and pop). These three instructions function like FLD, FST, and FSTP except the data transferred are integer data. The coprocessor automatically converts the internal extended-precision data to integer data. The size of the data is determined by the way the label is defined with DW, DD, or DQ in the assembly language program.

*BCD Data Transfer Instructions.* Two instructions load or store BCD signed-integer data. The FBLD instruction loads the top of the stack with BCD memory data and the FBSTP instruction stores the top of the stack and does a pop.

Example 8–7 shows how the assembler automatically adjusts the FLD, FILD, and FBLD instructions for different size operands. (Look closely at the machine coded forms of the instructions.) Note that this example begins with the .286 and .287 directives, which identify the microprocessor as an 80286 and the coprocessor as an 80287. If the 80386 microprocessor is in use with its coprocessor the directives .386 and .387 appear. The assembler assumes by default that the software is assembled for an 8086/8088 with an 8087 coprocessor.

**EXAMPLE 8–7**

```
                                .286
                                .287

0000                            DATAS   SEGMENT

0000 41F00000                   DATA1   DD    30.0          ;single-precision
0004 4059100000000000           DATA1   DQ    100.25        ;double-precision
000C 400487F34D6A161E4F76       DATA3   DT    33.9876       ;extended-precision

0016 001E                       DATA4   DW    30            ;16-bit integer
0018 0000001E                   DATA5   DD    30            ;32-bit integer
001C 000000000000001E           DATA6   DQ    30            ;64-bit integer

0024 00000000000000000030       DATA7   DT    30H           ;BCD 30

002E                            DATAS   ENDS

0000                            CODE    SEGMENT 'CODE'

                                        ASSUME   CS:CODE,DS:DATAS

0000 D9 06 0000 R                       FLD    DATA1
0004 DD 06 0004 R                       FLD    DATA2
0008 DB 2E 000C R                       FLD    DATA3

000C DF 06 0016 R                       FILD   DATA4
0010 DB 06 0018 R                       FILD   DATA5
0014 DF 2E 001C R                       FILD   DATA6

0018 DF 26 0024 R                       FBLD   DATA7

001C                            CODE    ENDS
                                        END
```

## Arithmetic Instructions

Arithmetic instructions for the coprocessor include: addition, subtraction, multiplication, division, and square root. The arithmetic-related instructions are: scaling, rounding, absolute value, and changing the sign.

Table 8–4 shows the basic addressing modes allowed for the arithmetic operations. Each addressing mode is shown with an example using the FADD (real addition) instruction. All arithmetic operations are floating-point operations except when memory data are referenced as an operand.

The *classic stack form* of addressing operand data (stack addressing) uses the top-of-the-stack as source operand and the next-to-the-top-of-the-stack as the destination operand. Afterward a pop removes the source datum from the stack and only the result in the destination register remains at the top of the stack. To use this addressing mode, the

**TABLE 8–4**
Arithmetic addressing modes

| *Mode* | *Form* | *Examples* |
|---|---|---|
| Stack | ST,ST(1) | FADD |
| Register | ST,ST(n) | FADD ST,ST(2) |
| | ST(n),ST | FADD ST(6),ST |
| Register pop | ST(n),ST | FADDP ST(3),ST |
| Memory | Operand | FADD DATA |

*Notes:* Stack addressing is fixed as ST,ST(1) and includes a pop so only the result remains at the top of the stack. Register addressing for any instruction can use a destination of ST or ST(n) as illustrated. n = register number 0–7.

instruction is placed in the program without any operands, such as FADD or FSUB. The FADD instruction adds ST to ST(1) and stores the answer at the top of the stack. It also removes the original two datum from the stack by popping. The FSUB instruction subtracts ST from ST(1) and leaves the difference at ST. Therefore, FSUBR subtracts ST(1) from ST and leaves the difference at ST.

The register-addressing mode uses ST for the top of the stack and ST(n) for another location, where n is the register number. With this form, one operand must be ST and the other is ST(n). To double the top of the stack, we use the FADD ST,ST(0) instruction, where ST(0) also addresses the top of the stack. One of the two operands in the register-addressing mode must be ST and the other must be in the form ST(n), where n is a stack register 0–7. For many instructions, either ST or ST(n) can be the destination. It is fairly important that the top of the stack is ST(0). This is accomplished by resetting or initializing the coprocessor before using it in a program. Another example of register addressing is FADD ST(1),ST, where the contents of ST are added to ST(1) and the result is placed into ST(1).

The top of the stack is always used as the destination for the memory-addressing mode because the coprocessor is a *stack-oriented* machine. For example, the FADD DATA instruction adds the real number contents of memory location DATA to the top of the stack.

*Arithmetic Operations.* The letter P in an opcode specifies a register pop after the operation (FADDP compared to FADD). The letter R in an opcode (subtraction and division only) indicates reverse mode. The reverse mode is useful for memory data because normally memory data subtract from the top of the stack. A reversed subtract instruction subtracts the top of the stack from memory and stores the result in the top of the stack. For example, if the top of the stack contains a 10 and memory location DATA1 contains a 1, the FSUB DATA1 instruction results in a +9 on the stack top and the FSUBR instruction results in a –9. Another example is FSUBR ST,ST(1), which will subtract ST from ST(1) and store the result on ST. A variant is FSUBR ST(1),ST, which will subtract ST(1) from ST and store the result on ST(1).

The letter I as the second letter in an opcode indicates that the memory operand is an integer. For example, the FADD DATA instruction is a floating-point addition, while the FIADD DATA instruction is an integer addition that adds the integer at memory location DATA to the floating-point number at the top of the stack. The same rules apply to FADD, FSUB, FMUL, and FDIV instructions.

*Arithmetic-Related Operations.* Other operations that are arithmetic in nature include: FSQRT (*square root*), FSCALE (*scale a number*), FPREM (*find partial remainder*), FRNDINT (*round to integer*), FXTRACT (*extract exponent and significand*), FABS (*find absolute value*), and FCHG (*change sign*). These instructions and the functions they perform follow:

| | |
|---|---|
| FSQRT | Finds the square root of the top of the stack and leaves the resultant square root at the top of the stack. An invalid error occurs for the square root of a negative number. For this reason, the IE bit of the status register should be tested whenever an invalid result can occur. The IE bit can be tested by loading the status register to AX with the FSTSW AX instruction, followed by TEST AX,1 to test the IE status bit. |
| FSCALE | Adds the contents of ST(1) (interpreted as an integer) to the exponent at the top of the stack. FSCALE can multiply or divide rapidly by powers of two. The value in ST(1) must be between $2^{-15}$ and $2^{+15}$. |
| FPREM | Performs modulo division of ST by ST(1). The resultant remainder is found in the top of the stack and has the same sign as the original dividend. Note that a modulo division results in a remainder without a quotient. |
| FRNDINT | Rounds the top of the stack to an integer. |
| FXTRACT | Decomposes the number at the top of the stack into two separate numbers that represent the value of the exponent and the value of the significand. The extracted significand is found at the top of the stack and the exponent is found at ST(1). We often use this instruction in converting a floating-point number into a form that can be printed. |
| FABS | Changes the sign of the top of the stack to positive. |
| FCHS | Changes the sign from positive to negative or negative to positive. |

## Comparison Instructions

The comparison instructions all examine data at the top of the stack in relation to another element and return the result of the comparison to the status-register condition code bits C3–C0. Comparisons that are allowed by the coprocessor are: FCOM (*floating-point compare*), FCOMP (*floating-point compare with a pop*), FCOMPP (*floating-point compare with 2 pops*), FICOM (*integer compare*), FICOMP (*integer compare and pop*), FSTS (*test*), and FXAM (*examine*). A list of these instructions follow with a description of their functions:

| | |
|---|---|
| FCOM | Compares the floating-point data at the top of the stack with an operand, which may be any register or any memory operand. If the operand is not coded with the instruction, the next stack element ST(1) is compared with the stack top ST. |
| FCOMP and FCOMPP | Both instructions perform as FCOM, but they also pop one or two datum from the stack. |
| FICOM and FICOMP | The top of the stack is compared with the integer stored at a memory operand. In addition to the compare, FICOMP also pops the top of the stack. |

| | |
|---|---|
| FTST | Tests the contents of the top of the stack against a zero. The result of the comparison is coded in the status-register condition code bits, as illustrated in Table 8–2, with the status register. See Table 8–3 for a way of using SAHF and the conditional jump instruction with FTST. |
| FXAM | Examines the stack top and modifies the condition code bits to indicate whether the contents are positive, negative, normalized, etc. Refer to the status register in Table 8–2. |

*Transcendental Operations* The transcendental instructions include: FPTAN (*partial tangent*), FPATAM (*partial arctangent*), F2XM1 ($2^x - 1$), FYL2X ($Y \log_2 X$), and FYL2XP1 ($Y \log_2(X + 1)$). In the 80387, 80486, and Pentium coprocessors, FSIN, FCOS, and FSINCOS are also allowed. A list of these operations follows with a description of each transcendental operation:

| | |
|---|---|
| FPTAN | Finds the partial tangent of $Y/X = \tan \theta$. The value of $\theta$ is at the top of the stack and must be between 0 and $\pi/4$. The result is a ratio found as ST = X and ST(1) = Y. If the value is outside of the allowable range, an invalid error occurs as indicated by the status-register IE bit. |
| FPATAN | Finds the partial arctangent as $\theta$ = ARCTAN X/Y. The value of X is at the top of the stack and Y is at ST(1). The values of X and Y must be as follows: $0 \leq Y < X < \infty$. The instruction pops the stack and leaves $\theta$ at the top of the stack. |
| F2AM1 | Finds the function $2^x - 1$. The value of X is taken from the top of the stack and the result is returned to the top of the stack. To obtain $2^x$ add one to the result at the top of the stack. This function can be used to derive the function listed in Table 8–5. Note that the constants $\log_2 10$ and $\log_2 e$ are built in as standard values for the coprocessor. |
| FYL2X | Finds $Y \log_2 X$. The value X is taken from the stack top and Y is taken from ST(1). The result is found at the top of the stack after a pop. The value of X must range between 0 and $\infty$ and the value of Y must be between $-\infty$ and $+\infty$. |
| FYL2XP1 | Finds $Y \log_2(X + 1)$. The value of X is taken from the stack top and Y is taken from ST(1). The result is found at the top of the stack after a pop. The value of X must range between 0 and $1 - \sqrt{2}/2$ and the value of Y must be between $-\infty$ and $+\infty$. |

**TABLE 8–5**
Exponential functions

| *Function* | *Equation* |
|---|---|
| $10^X$ | $2^X \log 2^{10}$ |
| $e^X$ | $2^X \log 2^e$ |
| $Y^X$ | $2^X \log 2^Y$ |

FSIN and FCOS — Finds the SIN or COS of ST and leaves the result at the top of the stack.

FSINCOS — Finds the SIN and COS of ST. Leaves the SIN at ST(0) and the COS at ST(1).

## Constant Operations

The coprocessor instruction set includes opcodes that return constants to the top of the stack. A list of these instructions appears in Table 8–6.

## Coprocessor Control Instructions

The coprocessor has control instructions for initialization, exception handling, and task switching. The control instructions have two forms. Both FNINIT and FINIT initialize the coprocessor. The difference is that FNINIT does not cause any wait states while FINIT does. The microprocessor waits for the FINIT instruction by testing the $\overline{\text{BUSY}}$ pin on the coprocessor. All control instructions have these two forms. Following is a list of each control instruction with its function:

FINIT/FNINIT — This instruction performs a reset operation on the arithmetic coprocessor (refer to Table 8–7 for the reset conditions). The coprocessor operates with a closure of projective (unsigned) infinity, rounds to the nearest number or even number, and uses extended precision when reset or initialized. It also sets register 0 as the top of the stack.

FSETPM — Changes the addressing mode of the coprocessor to the protected addressing mode. This mode is used when the microprocessor is also operated in the protected mode. Protected mode can only be exited by a hardware reset, or, in the case of the 80386/80486, with a change to the control register.

FLDCW — Loads the control register with the word addressed by the operand.

**TABLE 8–6**
Constant operations

| *Instruction* | *Constant Pushed to ST* |
|---|---|
| FLDZ | + 0.0 |
| FLD1 | + 1.0 |
| FLDPI | $\pi$ |
| FLDL2T | $\log 2^{10}$ |
| FLDL2E | $\log 2^{e}$ |
| FLDLG2 | $\log 10^{2}$ |
| FLDLN2 | $\log e^{2}$ |

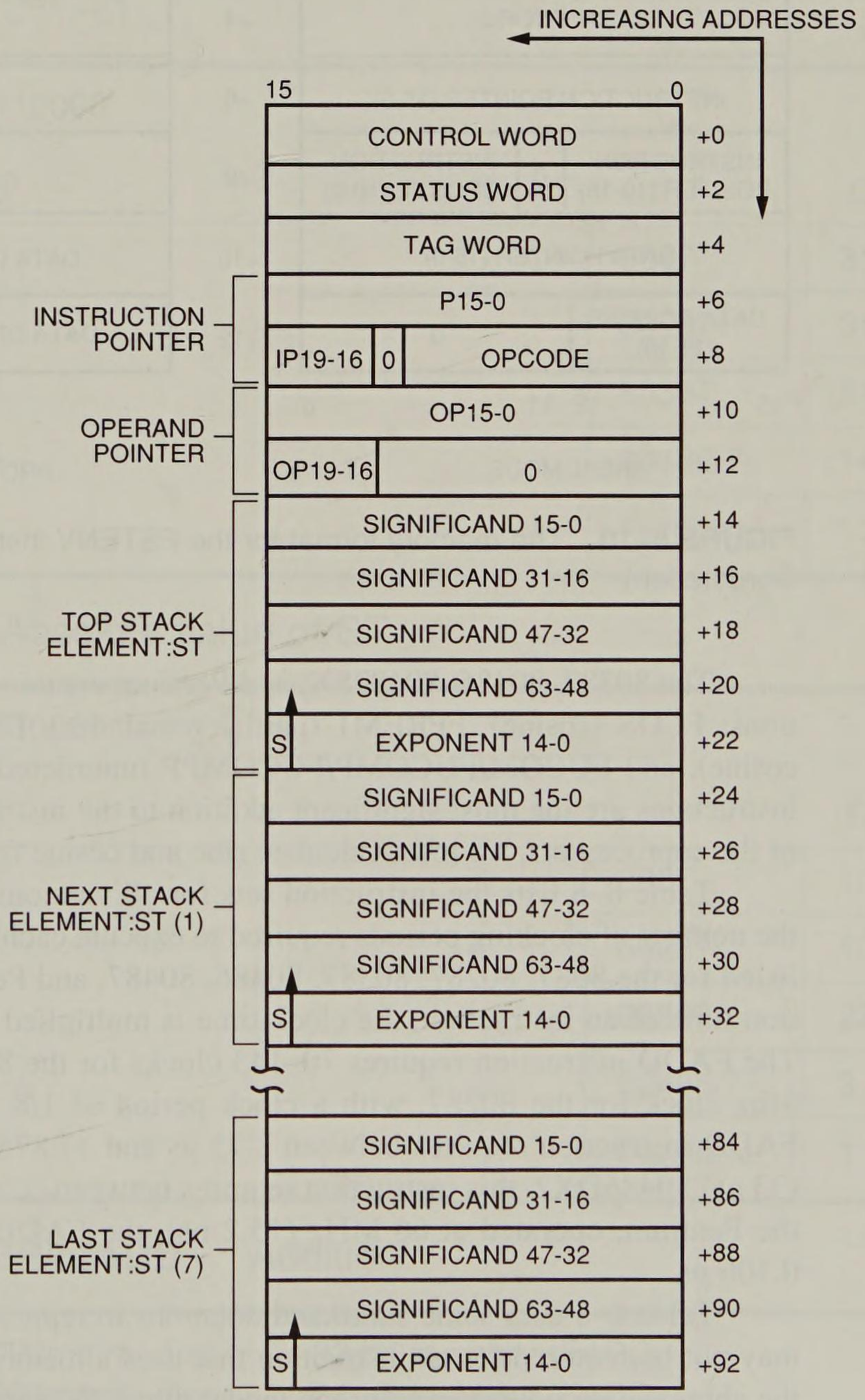
**TABLE 8–7** Coprocessor state after a reset or initialization

| *Field* | *Value* | *Condition* |
|---|---|---|
| Infinity | 0 | Projective |
| Rounding | 00 | Round to nearest number |
| Precision | 11 | Extended precision |
| Error Masks | 11111 | Error bits disabled |
| Busy | 0 | Not busy |
| C3–C0 | ???? | Unknown |
| TOP | 000 | Register 000 |
| ES | 0 | No error |
| Error Bits | 00000 | No errors |
| All tags | 11 | Empty |
| Registers | – | Not changed |

| | |
|---|---|
| FSTCW/FNSTCW | Stores the control register into the word-sized memory operand. |
| FSTSW AX/FNSTSW AX | Copies the contents of the control register to the AX register. |
| FCLEX/FNCLEX | Clears the error flags in the status register and also the busy flag. |
| FSAVE/FNSAVE | Writes the entire state of the machine to memory. Figure 8–9 shows the memory layout for this instruction. |
| FRSTOR | Restores the state of the machine from memory. This instruction is used to restore the information saved by FSAVE/FNSAVE. |
| FSTENV/FNSTENV | Stores the environment of the coprocessor as shown in Figure 8–10. |
| FLDENV | Reloads the environment saved by FSTENV/FNSTENV. |
| FINCST | Increments the stack pointer. |
| FDECSTP | Decrements the stack pointer. |
| FFREE | Frees a register by changing the destination register's tag to empty. It does not affect the contents of the register. |
| FNOP | Floating-point coprocessor NOP. |
| FWAIT | Causes the microprocessor to wait for the coprocessor to finish an operation. FWAIT should be used before the microprocessor accesses memory data that are affected by the coprocessor. |

## COPROCESSOR INSTRUCTIONS

Although we have yet to talk about the microprocessor circuitry, we can discuss the instruction sets of these coprocessors and the differences between them and other coprocessors. These newer coprocessors contain the same basic instructions provided by the earlier versions with a few additional instructions.




NOTES:
S = Sign
Bit 0 of each field is right-most, least-significant bit of corresponding register field.
Bit 63 of significand is integer bit (assumed binary point is immediately to the right).

**FIGURE 8–9** Memory format when the 80X87 registers are stored by the FSAVE instruction. (Courtesy of Intel Corporation)

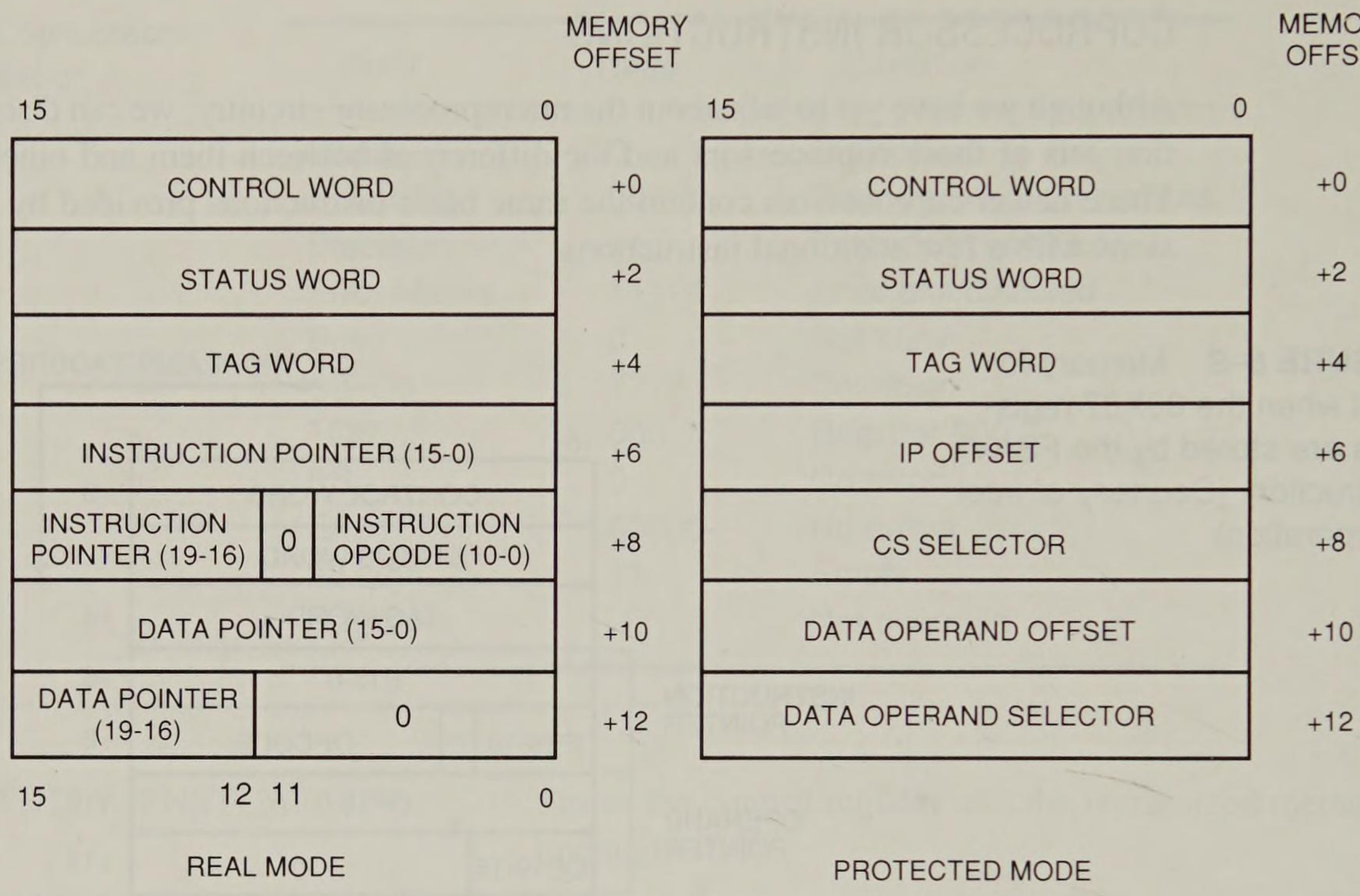



**FIGURE 8–10** The memory format for the FSTENV instruction. (Courtesy of Intel Corporation)

The 80387, 80486, 80487SX, and Pentium contain the following additional instructions: FCOS (cosine), FPREM1 (partial remainder), FSIN (sine), FSINCOS (sine and cosine), and FUCOM/FUCOMP/FUCOMPP (unordered compare). The sine and cosine instructions are the most significant addition to the instruction set. In the earlier versions of the coprocessors, we must calculate sine and cosine from the tangent.

Table 8–8 lists the instruction sets for all versions of the coprocessor. It also lists the number of clocking periods required to execute each instruction. Execution times are listed for the 8087, 80287, 80387, 80486, 80487, and Pentium. To determine the execution time of an instruction, the clock time is multiplied times the listed execution time. The FADD instruction requires 70–143 clocks for the 80287. Suppose that we use an 8 Mhz clock for the 80287, with a clock period of 1/8 MHz or 125 ns per clock. The FADD instruction requires between 8.75 μs and 17.875 μs to execute. Using a 33 Mhz (33 ns) 80486DX2, this instruction requires between 0.264 μs and 0.66 μs to execute. On the Pentium, operated at 66 MHz (15.2 ns), the FADD requires beteen 0.0152 μs and 0.106 μs.

Table 8–8 uses some shorthand notations to represent the displacement that may or may not be required for an instruction that uses a memory addressing mode. It also uses the abbreviation oo to represent the mode; mmm to represent a register/memory addressing mode; and rrr to represent one of the floating-point coprocessor registers ST(0)–ST(7). The d (destination) bit that appears in some instruction opcodes defines the direction of the data flow as in FADD ST,ST(2) or FADD ST(2),ST. The d bit is a logic 0 for flow toward ST as in FADD ST,ST(2) where ST holds the sum after the addition, and a logic 1 for FADD ST(2),ST where ST(2) holds the sum.

Some instructions allow a choice of whether a wait is inserted. For example, the FSTSW AX instruction copies the status register into AX. The FNSTSW AX instruction also copies the status register to AX, but without a wait.

**TABLE 8–8** The instruction set of the arithmetic coprocessor (pp. 293–312)

**F2XM1** $2^{ST} - 1$

11011001 11110000

| Example | | Clocks |
|---|---|---|
| F2XM1 | 8087 | 310–630 |
| | 80287 | 310–630 |
| | 80387 | 211–476 |
| | 80486/7 | 140–279 |
| | Pentium | 13–57 |

**FABS** Absolute value of ST

11011001 11100001

| Example | | Clocks |
|---|---|---|
| FABS | 8087 | 10–17 |
| | 80287 | 10–17 |
| | 80387 | 22 |
| | 80486/7 | 3 |
| | Pentium | 1 |

**FADD/FADDP/FIADD** Addition

| | |
|---|---|
| 11011000 oo000mmm disp | 32-bit memory (FADD) |
| 11011100 oo000mmm disp | 64-bit memory (FADD) |
| 11011d00 11000rrr | FADD ST,ST(rrr) |
| 11011110 11000rrr | FADDP ST,ST(rrr) |
| 11011110 oo000mmm disp | 16-bit memory (FIADD) |
| 11011010 oo000mmm disp | 32-bit memory (FIADD) |

| Format | Examples | | Clocks |
|---|---|---|---|
| FADD<br>FADDP<br>FIADD | FADD DATA<br>FADD ST,ST(1)<br>FADDP<br>FIADD NUMBER<br>FADD ST,ST(3)<br>FADDP ST,ST(2)<br>FADD ST(2),ST | 8087 | 70–143 |
| | | 80287 | 70–143 |
| | | 80387 | 23–72 |
| | | 80486/7 | 8–20 |
| | | Pentium | 1–7 |

**FCLEX/FNCLEX** Clear errors

11011011 11100010

| Example | | Clocks |
|---|---|---|
| FCLEX<br>FNCLEX | 8087 | 2–8 |
| | 80287 | 2–8 |
| | 80387 | 11 |
| | 80486/7 | 7 |
| | Pentium | 9 |

**FCOM/FCOMP/FCOMPP/FICOM/FICOMP** Compare

| | |
|---|---|
| 11011000 oo010mmm disp | 32-bit memory (FCOM) |
| 11011100 oo010mmm disp | 64-bit memory (FCOM) |
| 11011000 11010rrr | FCOM ST(rrr) |
| 11011000 oo011mmm disp | 32-bit memory (FCOMP) |
| 11011100 oo011mmm disp | 64-bit memory (FCOMP) |
| 11011000 11011rrr | FCOMP ST(rrr) |
| 11011110 11011001 | FCOMPP |
| 11011110 oo010mmm disp | 16-bit memory (FICOM) |
| 11011010 oo010mmm disp | 32-bit memory (FICOM) |
| 11011110 oo011mmm disp | 16-bit memory (FICOMP) |
| 11011010 oo011mmm disp | 32-bit memory (FICOMP) |

| Format | Examples | | |
|---|---|---|---|
| FCOM<br>FCOMP<br>FCOMPP<br>FICOM<br>FICOMP | FCOM ST(2)<br>FCOMP DATA<br>FCOMPP<br>FICOM NUMBER<br>FICOMP DATA3 | 8087 | 40–93 |
| | | 80287 | 40–93 |
| | | 80387 | 24–63 |
| | | 80486/7 | 15–20 |
| | | Pentium | 4–8 |

**FCOS** Cosine of ST

11011001 11111111

| Example | | Clocks |
|---|---|---|
| FCOS | 8087 | — |
| | 80287 | — |
| | 80387 | 123–772 |
| | 80486/7 | 193–279 |
| | Pentium | 18–124 |

**FDECSTP** Decrement stack pointer

11011001 11110110

| Example | | Clocks |
|---|---|---|
| FDECSTP | 8087 | 6–12 |
| | 80287 | 6–12 |
| | 80387 | 22 |
| | 80486/7 | 3 |
| | Pentium | 1 |

## FDISI/FNDISI Disable interrupts

11011011 11100001

(ignored on the 80287, 80387, 80486/7, and Pentium)

| Example | | Clocks |
|---|---|---|
| FDISI<br>FNDISI | 8087 | 2–8 |
| | 80287 | — |
| | 80387 | — |
| | 80486/7 | — |
| | Pentium | — |

## FDIV/FDIVP/FIDIV Division

```
11011000  oo110mmm  disp      32-bit memory (FDIV)
11011100  oo100mmm  disp      64-bit memory (FDIV)
11011d00  11111rrr            FDIV ST,ST(rrr)
11011110  11111rrr            FDIVP ST,ST(rrr)
11011110  oo110mmm  disp      16-bit memory (FIDIV)
11011010  oo110mmm  disp      32-bit memory (FIDIV)
```

| Format | Examples | | Clocks |
|---|---|---|---|
| FDIV<br>FDIVP<br>FIDIV | FDIV DATA<br>FDIV ST,ST(3)<br>FDIVP<br>FIDIVNUMBER<br>FDIV ST,ST(5)<br>FDIVP ST,ST(2)<br>FDIV ST(2),ST | 8087 | 191–243 |
| | | 80287 | 191–243 |
| | | 80387 | 88–140 |
| | | 80486/7 | 8–89 |
| | | Pentium | 39–42 |

## FDIVR/FDIVRP/FIDIVR Division reversed

```
11011000  oo111mmm  disp      32-bit memory (FDIVR)
11011100  oo111mmm  disp      64-bit memory (FDIVR)
11011d00  11110rrr            FDIVR ST,ST(rrr)
11011110  11110rrr            FDIVRP ST,ST(rrr)
11011110  oo111mmm  disp      16-bit memory (FIDIVR)
11011010  oo111mmm  disp      32-bit memory (FIDIVR)
```

| Format | Examples | | |
|---|---|---|---|
| FDIVR<br>FDIVRP<br>FIDIVR | FDIVR DATA<br>FDIVR ST,ST(3)<br>FDIVRP<br>FIDIVR NUMBER<br>FDIVR ST,ST(5)<br>FDIVRP ST,ST(2)<br>FDIVR ST(2),ST | 8087 | 191–243 |
| | | 80287 | 191–243 |
| | | 80387 | 88–140 |
| | | 80486/7 | 8–89 |
| | | Pentium | 39–42 |

## FENI/FNENI Disable interrupts

11011011 11100000

(ignored on the 80287, 80387, 80486/7 and Pentium)

| Example | | |
|---|---|---|
| FENI<br>FNENI | 8087 | 2–8 |
| | 80287 | — |
| | 80387 | — |
| | 80486/7 | — |
| | Pentium | — |

**FFREE** Free register

11011101 11000rrr

| Format | Examples | | Clocks |
|---|---|---|---|
| FFREE | FFREE<br>FFREE ST(1)<br>FFREE ST(2) | 8087 | 9–16 |
| | | 80287 | 9–16 |
| | | 80387 | 18 |
| | | 80486/7 | 3 |
| | | Pentium | 1 |

**FINCSTP** Increment stack pointer

11011001 11110111

| Example | | Clocks |
|---|---|---|
| FINCSTP | 8087 | 6–12 |
| | 80287 | 6–12 |
| | 80387 | 21 |
| | 80486/7 | 3 |
| | Pentium | 1 |

**FINIT/FNINIT** Initialize coprocessor

11011001 11110110

| Example | | Clocks |
|---|---|---|
| FINIT<br>FNINIT | 8087 | 2–8 |
| | 80287 | 2–8 |
| | 80387 | 33 |
| | 80486/7 | 17 |
| | Pentium | 12–16 |

## FLD/FILD/FBLD Load data to ST(0)

```
11011001  oo000mmm  disp      32-bit memory (FLD)
11011101  oo000mmm  disp      64-bit memory (FLD)
11011011  oo101mmm  disp      80-bit memory (FLD)
11011111  oo000mmm  disp      16-bit memory (FILD)
11011011  oo000mmm  disp      32-bit memory (FILD)
11011111  oo101mmm  disp      64-bit memory (FILD)
11011111  oo100mmm  disp      80-bit memory (FBLD)
```

| Format | Examples | | Clocks |
|---|---|---|---|
| FLD<br>FILD<br>FBLD | FLD DATA<br>FILD DATA1<br>FBLD DEC_DATA | 8087 | 17–310 |
| | | 80287 | 17–310 |
| | | 80387 | 14–275 |
| | | 80486/7 | 3–103 |
| | | Pentium | 1–3 |

## FLD1 Load + 1.0 to ST(0)

11011001 11101000

| Example | | Clocks |
|---|---|---|
| FLD1 | 8087 | 15–21 |
| | 80287 | 15–21 |
| | 80387 | 24 |
| | 80486/7 | 4 |
| | Pentium | 2 |

**FLDZ** Load + 0.0 to ST(0)

11011001 11101110

| Example | | Clocks |
|---|---|---|
| FLDZ | 8087 | 11–17 |
| | 80287 | 11–17 |
| | 80387 | 20 |
| | 80486/7 | 4 |
| | Pentium | 2 |

**FLDPI** Load $\pi$ to ST(0)

11011001 11101011

| Example | | Clocks |
|---|---|---|
| FLDPI | 8087 | 16–22 |
| | 80287 | 16–22 |
| | 80387 | 40 |
| | 80486/7 | 8 |
| | Pentium | 3–5 |

**FLDL2E** Load $\log_2 e$ to ST(0)

11011001 11101010

| Example | | Clocks |
|---|---|---|
| FLDL2E | 8087 | 15–21 |
| | 80287 | 15–21 |
| | 80387 | 40 |
| | 80486/7 | 8 |
| | Pentium | 3–5 |

**FLDL2T** Load $\log_2 10$ to ST(0)

11011001 11101001

| Example | | Clocks |
|---|---|---|
| FLDL2T | 8087 | 16–22 |
| | 80287 | 16–22 |
| | 80387 | 40 |
| | 80486/7 | 8 |
| | Pentium | 3–5 |

**FLDLG2** Load $\log_{10} 2$ to ST(0)

11011001 11101000

| Example | | Clocks |
|---|---|---|
| FLDLG2 | 8087 | 18–24 |
| | 80287 | 18–24 |
| | 80387 | 41 |
| | 80486/7 | 8 |
| | Pentium | 3–5 |

**FLDLN2** Load $\log_e 2$ to ST(0)

11011001 11101101

| Example | | Clocks |
|---|---|---|
| FLDLN2 | 8087 | 17–23 |
| | 80287 | 17–23 |
| | 80387 | 41 |
| | 80486/7 | 8 |
| | Pentium | 3–5 |

**FLDCW** Load control register

11011001 oo101mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FLDCW | FLDCW DATA<br>FLDCW STATUS | 8087 | 7–14 |
| | | 80287 | 7–14 |
| | | 80387 | 19 |
| | | 80486/7 | 4 |
| | | Pentium | 7 |

**FLDENV** Load environment

11011001 oo100mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FLDENV | FLDENV ENVIRON<br>FLDENV DATA | 8087 | 35–45 |
| | | 80287 | 25–45 |
| | | 80387 | 71 |
| | | 80486/7 | 34–44 |
| | | Pentium | 32–37 |

**FMUL/FMULP/FIMUL** Multiplication

| | |
|---|---|
| 11011000 oo001mmm disp | 32-bit memory (FMUL) |
| 11011100 oo001mmm disp | 64-bit memory (FMUL) |
| 11011d00 11001rrr | FMUL ST,ST(rrr) |
| 11011110 11001rrr | FMULP ST,ST(rrr) |
| 11011110 oo001mmm disp | 16-bit memory (FIMUL) |
| 11011010 oo001mmm disp | 32-bit memory (FIMUL) |

| Format | Examples | | Clocks |
|---|---|---|---|
| FMUL<br>FMULP<br>FIMUL | FMUL DATA<br>FMUL ST,ST(2)<br>FMUL ST(2),ST<br>FMULP<br>FIMUL DATA3 | 8087 | 110–168 |
| | | 80287 | 110–168 |
| | | 80387 | 29–82 |
| | | 80486/7 | 11–27 |
| | | Pentium | 1–7 |

**FNOP** No operation

11011001 11010000

| Example | | Clocks |
|---|---|---|
| FNOP | 8087 | 10–16 |
| | 80287 | 10–16 |
| | 80387 | 12 |
| | 80486/7 | 3 |
| | Pentium | 1 |

**FPATAN** Partial arctangent of ST(0)

11011001 11110011

| Example | | Clocks |
|---|---|---|
| FPATAN | 8087 | 250–800 |
| | 80287 | 250–800 |
| | 80387 | 314–487 |
| | 80486/7 | 218–303 |
| | Pentium | 19–134 |

**FPREM** Partial remainder

11011001 11111000

| Example | | Clocks |
|---|---|---|
| FPREM | 8087 | 15–190 |
| | 80287 | 15–190 |
| | 80387 | 74–155 |
| | 80486/7 | 70–138 |
| | Pentium | 16–64 |

**FPREM1** Partial remainder (IEEE)

11011001 11110101

| Example | | Clocks |
|---|---|---|
| FPREM1 | 8087 | — |
| | 80287 | — |
| | 80387 | 95–185 |
| | 80486/7 | 72–167 |
| | Pentium | 20–70 |

**FPTAN** Partial tangent of ST(0)

11011001 11110010

| Example | | Clocks |
|---|---|---|
| FPTAN | 8087 | 30–450 |
| | 80287 | 30–450 |
| | 80387 | 191–497 |
| | 80486/7 | 200–273 |
| | Pentium | 17–173 |

**FRNDINT** Round ST(0) to an integer

11011001 11111100

| Example | | Clocks |
|---|---|---|
| FRNDINT | 8087 | 16–50 |
| | 80287 | 16–50 |
| | 80387 | 66–80 |
| | 80486/7 | 21–30 |
| | Pentium | 9–20 |

**FRSTOR** Restore state

11011101 oo110mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FRSTOR | FRSTOR DATA<br>FRSTOR STATE<br>FRSTOR MACHINE | 8087 | 197–207 |
| | | 80287 | 197–207 |
| | | 80387 | 308 |
| | | 80486/7 | 120–131 |
| | | Pentium | 70–95 |

**FSAVE/FNSAVE** Save machine state

11011101 oo110mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FSAVE<br>FNSAVE | FSAVE STATE<br>FNSAVE STATUS<br>FSAVE MACHINE | 8087 | 197–207 |
| | | 80287 | 197–207 |
| | | 80387 | 375 |
| | | 80486/7 | 143–154 |
| | | Pentium | 124–151 |

**FSCALE** Scale ST(0) by ST(1)

11011001 11111101

| Example | | Clocks |
|---|---|---|
| FSCALE | 8087 | 32–38 |
| | 80287 | 32–38 |
| | 80387 | 67–86 |
| | 80486/7 | 30–32 |
| | Pentium | 20–31 |

**FSETPM** Set protected mode

11011011 11100100

| Example | | Clocks |
|---|---|---|
| FSETPM | 8087 | — |
| | 80287 | 2–18 |
| | 80387 | 12 |
| | 80486/7 | — |
| | Pentium | — |

**FSIN** Sine of ST(0)

11011001 11111110

| Example | | Clocks |
|---|---|---|
| FSIN | 8087 | — |
| | 80287 | — |
| | 80387 | 122–771 |
| | 80486/7 | 193–279 |
| | Pentium | 16–126 |

**FSINCOS** Find sine and cosine of ST(0)

11011001 11111011

| Example | | Clocks |
|---|---|---|
| FSINCOS | 8087 | — |
| | 80287 | — |
| | 80387 | 194–809 |
| | 80486/7 | 243–329 |
| | Pentium | 17–137 |

**FSQRT** Square root of ST(0)

11011001 11111010

| Example | | Clocks |
|---|---|---|
| FSQRT | 8087 | 180–186 |
| | 80287 | 180–186 |
| | 80387 | 122–129 |
| | 80486/7 | 83–87 |
| | Pentium | 70 |

**FST/FSTP/FIST/FISTP/FBSTP** Store

| | |
|---|---|
| 11011001 oo010mmm disp | 32-bit memory (FST) |
| 11011101 oo010mmm disp | 64-bit memory (FST) |
| 11011101 11010rrr | FST ST(rrr) |
| 11011011 oo011mmm disp | 32-bit memory (FSTP) |
| 11011101 oo011mmm disp | 64-bit memory (FSTP) |
| 11011011 oo111mmm disp | 80-bit memory (FSTP) |
| 11011101 11001rrr | FSTP ST(rrr) |
| 11011111 oo010mmm disp | 16-bit memory (FIST) |
| 11011011 oo010mmm disp | 32-bit memory (FIST) |
| 11011111 oo011mmm disp | 16-bit memory (FISTP) |
| 11011011 oo011mmm disp | 32-bit memory (FISTP) |
| 11011111 oo111mmm disp | 64-bit memory (FISTP) |
| 11011111 oo110mmm disp | 80-bit memory (FBSTP) |

| Format | Examples | | Clocks |
|---|---|---|---|
| FST<br>FSTP<br>FIST<br>FISTP<br>FBSTP | FST DATA<br>FST ST(3)<br>FST<br>FSTP<br>FIST DATA2<br>FBSTP DATA6<br>FISTP DATA9 | 8087 | 15–540 |
| | | 80287 | 15–540 |
| | | 80387 | 11–534 |
| | | 80486/7 | 3–176 |
| | | Pentium | 1–3 |

**FSTCW/FNSTCW** Store control register

11011001 oo111mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FSTCW<br>FNSTCW | FSTCW CONTROL<br>FNSTCW STATUS<br>FSTCW MACHINE | 8087 | 12–18 |
| | | 80287 | 12–18 |
| | | 80387 | 15 |
| | | 80486/7 | 3 |
| | | Pentium | 2 |

**FSTENV/FNSTENV** Store environment

11011001 oo110mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FSTENV<br>FNSTENV | FSTENV CONTROL<br>FNSTENV STATUS<br>FSTENV MACHINE | 8087 | 40–50 |
| | | 80287 | 40–50 |
| | | 80387 | 103–104 |
| | | 80486/7 | 58–67 |
| | | Pentium | 48–50 |

## **FSTSW/FNSTSW** Store status register

11011101 oo111mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| FSTSW<br>FNSTSW | FSTSW CONTROL<br>FNSTSW STATUS<br>FSTSW MACHINE<br>FSTSW AX | 8087 | 12–18 |
| | | 80287 | 12–18 |
| | | 80387 | 15 |
| | | 80486/7 | 3 |
| | | Pentium | 2–5 |

## **FSUB/FSUBP/FISUB** Subtraction

| | |
|---|---|
| 11011000 oo100mmm disp | 32-bit memory (FSUB) |
| 11011100 oo100mmm disp | 64-bit memory (FSUB) |
| 11011d00 11101rrr | FSUB ST,ST(rrr) |
| 11011110 11101rrr | FSUBP ST,ST(rrr) |
| 11011110 oo100mmm disp | 16-bit memory (FISUB) |
| 11011010 oo100mmm disp | 32-bit memory (FISUB) |

| Format | Examples | | Clocks |
|---|---|---|---|
| FSUB<br>FSUBP<br>FISUB | FSUB DATA<br>FSUB ST,ST(2)<br>FSUB ST(2),ST<br>FSUBP<br>FISUB DATA3 | 8087 | 70–143 |
| | | 80287 | 70–143 |
| | | 80387 | 29–82 |
| | | 80486/7 | 8–35 |
| | | Pentium | 1–3 |

## **FSUBR/FSUBRP/FISUBR** Reverse subtraction

| | |
|---|---|
| 11011000 oo101mmm disp | 32-bit memory (FSUBR) |
| 11011100 oo101mmm disp | 64-bit memory (FSUBR) |
| 11011d00 11100rrr | FSUBR ST,ST(rrr) |
| 11011110 11100rrr | FSUBRP ST,ST(rrr) |
| 11011110 oo101mmm disp | 16-bit memory (FISUBR) |
| 11011010 oo101mmm disp | 32-bit memory (FISUBR) |

| Format | Examples | | Clocks |
|---|---|---|---|
| FSUBR<br>FSUBRP<br>FISUBR | FSUBR DATA<br>FSUBR ST,ST(2)<br>FSUBR ST(2),ST<br>FSUBRP<br>FISUBR DATA3 | 8087 | 70–143 |
| | | 80287 | 70–143 |
| | | 80387 | 29–82 |
| | | 80486/7 | 8–35 |
| | | Pentium | 1–3 |

**FTST** Compare ST(0) with + 0.0

11011001 11100100

| Example | | Clocks |
|---|---|---|
| FTST | 8087 | 38–48 |
| | 80287 | 38–48 |
| | 80387 | 28 |
| | 80486/7 | 4 |
| | Pentium | 1–4 |

**FUCOM/FUCOMP/FUCOMPP** Unordered compare

| | |
|---|---|
| 11011101 11100rrr | FUCOM ST,ST(rrr) |
| 11011101 11101rrr | FUCOMP ST,ST(rrr) |
| 11011101 11101001 | FUCOMPP |

| Format | Examples | | Clocks |
|---|---|---|---|
| FUCOM<br>FUCOMP<br>FUCOMPP | FUCOM ST,ST(2)<br>FUCOM<br>FUCOMP ST,ST(3)<br>FUCOMP<br>FUCOMPP | 8087 | — |
| | | 80287 | — |
| | | 80387 | 24–26 |
| | | 80486/7 | 4–5 |
| | | Pentium | 1–4 |

**FWAIT** Wait

10011011

| Example | | Clocks |
|---|---|---|
| FWAIT | 8087 | 4 |
| | 80287 | 3 |
| | 80387 | 6 |
| | 80486/7 | 1–3 |
| | Pentium | 1 |

**FXAM** Examine ST(0)

11011001 11100101

| Example | | Clocks |
|---|---|---|
| FXAM | 8087 | 12–23 |
| | 80287 | 12–23 |
| | 80387 | 30–38 |
| | 80486/7 | 8 |
| | Pentium | 21 |

**FXCH** Exchange ST(0) with another register

11011001 11001rrr FXCH ST,ST(rrr)

| Format | Examples | | Clocks |
|---|---|---|---|
| FXCH | FXCH ST,ST(1)<br>FXCH<br>FXCH ST,ST(4) | 8087 | 10–15 |
| | | 80287 | 10–15 |
| | | 80387 | 18 |
| | | 80486/7 | 4 |
| | | Pentium | 1 |

**FXTRACT** Extract components of ST(0)

11011001 11110100

| Example | | Clocks |
|---|---|---|
| FXTRACT | 8087 | 27–55 |
| | 80287 | 27–55 |
| | 80387 | 70–76 |
| | 80486/7 | 16–20 |
| | Pentium | 13 |

**FYL2X** $ST(1) \times \log_2 ST(0)$

11011001 11110001

| Example | | Clocks |
|---|---|---|
| FYL2X | 8087 | 900–1100 |
| | 80287 | 900–1100 |
| | 80387 | 120–538 |
| | 80486/7 | 196–329 |
| | Pentium | 22–111 |

**FXL2XP1** $ST(1) \times \log_2 [ST(0) + 1.0]$

11011001 11111001

| Example | | Clocks |
|---|---|---|
| FXL2XP1 | 8087 | 700–1000 |
| | 80287 | 700–1000 |
| | 80387 | 257–547 |
| | 80486/7 | 171–326 |
| | Pentium | 22–103 |

Notes: d = direction where d = 0 for ST as the destination, and d = 1 for ST as the source; rrr = floating-point register number; oo = mode; mmm = r/m field; and disp = displacement.

# 8–4 PROGRAMMING WITH THE ARITHMETIC COPROCESSOR

This section of the chapter provides programming examples for the arithmetic coprocessor. Each example is chosen to illustrate a programming technique for the coprocessor.

## Calculating the Area of a Circle

This programming example illustrates a simple method of addressing the coprocessor stack. First recall that the equation for calculating the area of a circle is $A = \pi R^2$. A procedure that performs this calculation is listed in Example 8–8.

**EXAMPLE 8–8**

```
                        ;Procedure that calculates the area of a circle.
                        ;
                        ;The radius must be stored at memory location RADIUS
                        ;before calling this procedure.  The result is found
                        ;in memory location AREA after the procedure.
                        ;
0000                    AREAS   PROC    FAR

0000 D9 06 0004 R               FLD     RADIUS          ;radius to ST
0004 D8 C8                      FMUL    ST,ST(0)        ;square radius
0006 D9 EB                      FLDPI                   ;n to ST
0008 DE C9                      FMUL                    ;multiply ST = ST x ST(1)
000A D9 1E 0000 R               FSTP    AREA            ;save area
000E 9B                         FWAIT                   ;wait for coprocessor
000F CB                         RET

0010                    AREAS   ENDP
```

Although this is a rather simple procedure, it does illustrate the operation of the stack. To provide a better understanding of the operation of the stack, Figure 8–11 shows the contents of the stack after each instruction of Example 8–8 executes.

The first instruction loads the contents of memory location RADIUS to the top of the stack. Next the FMUL ST,ST(0) instruction squares the radius on the top of the stack. The FLDPI instruction loads $\pi$ to the stack top. The FMUL instruction uses the classic stack-addressing mode to multiply ST by ST(1). After the multiplication, the prior values of ST and ST(1) are removed from the stack and the product replaces them at the top of the stack. Finally the FSTP instruction copies the top of the stack, the area, to memory location AREA and clears the stack.

The FWAIT instruction appears just before the return instruction in this example. The reason that we use the FWAIT instruction is to wait for the coprocessor to finish finding the area before returning. If we were not to wait, the main program might access memory location AREA before the coprocessor stores the result into location AREA. Also, notice how care was taken to remove all stack data. This is important because if data remain on the stack at the end of the procedure, the stack top will no longer be register 0. This could cause problems because software assumes that the top of the stack is register 0.

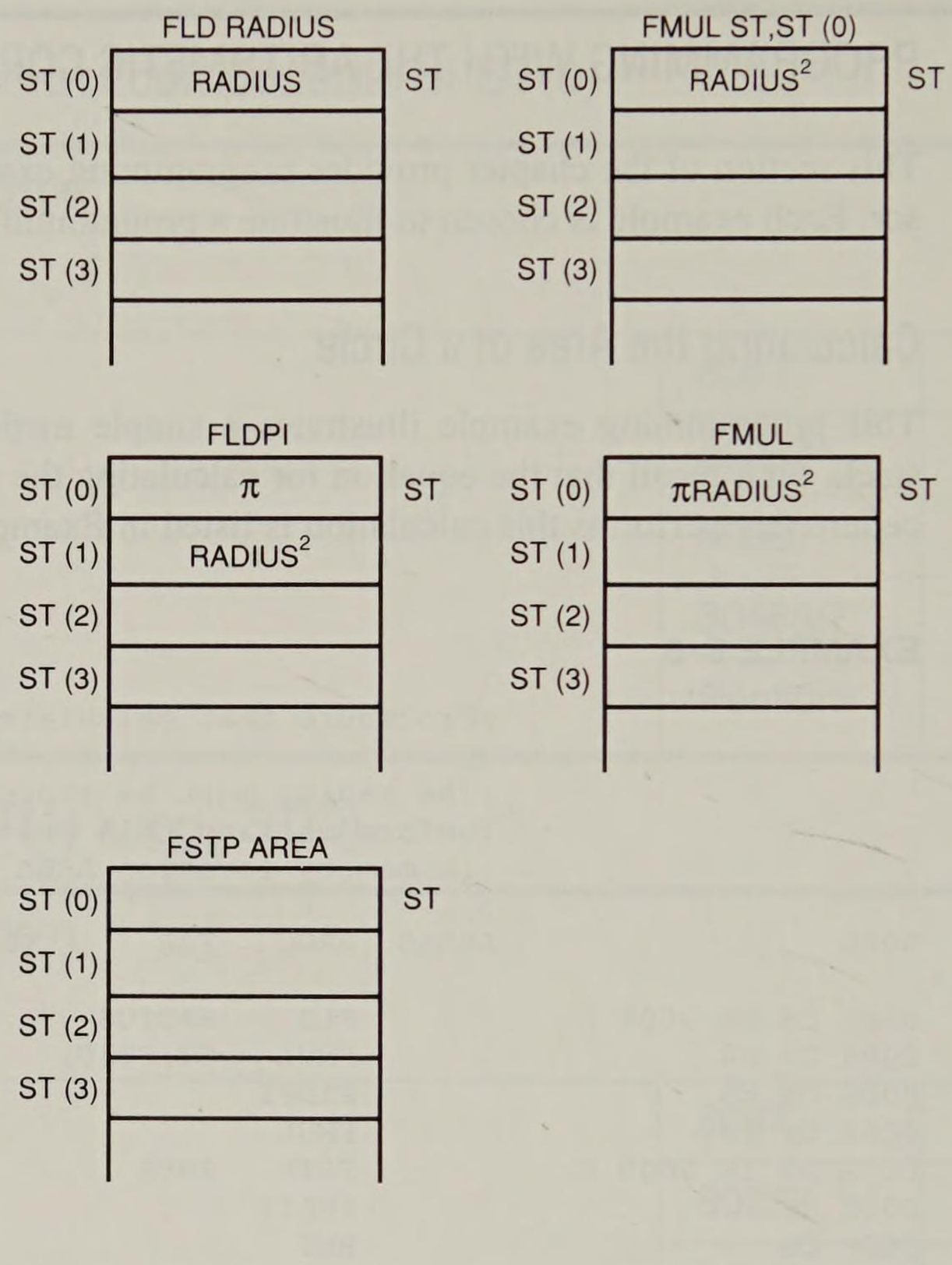



**FIGURE 8–11** Operation of the 80287 stack with the procedure of Example 12–6. Note: Stack shown after the execution of the indicated instruction.

## Finding the Resonant Frequency

An equation commonly used in electronics is the formula for determining the resonant frequency of an LC circuit. The equation solved by the procedure illustrated in Example 8–9 is $Fr = 1/(2\pi\sqrt{LC})$.

**EXAMPLE 8–9**

```
0000                    DATAS   SEGMENT

0000 00000000           RESO    DD    ?                 ;resonant frequency
0004 358637BD           L       DD    .000001           ;inductance
0008 358637BD           C       DD    .000001           ;capacitance
000C 40000000           TWO     DD    2.0               ;constant

0010                    DATAS   ENDS

0000                    CODE    SEGMENT USE16 'CODE'

                                ASSUME  CS:CODE,DS:DATAS

                        ;Procedure that finds the resonant frequency.

0000                    FREQ    PROC    FAR

0000 D9 06 0004 R               FLD  L                  ;get L
```

```
0004 D8 0E 0008 R          FMUL C              ;find LC

0008 D9 FA                 FSQRT               ;find √LC

000A D8 0E 000C R          FMUL TWO            ;find 2π√LC

000E D9 EB                 FLDPI               ;get n
0010 DE C9                 FMUL                ;get 2π√LC

0012 D9 E8                 FLD1                ;get 1
0014 DE F1                 FDIVR               ;form 1/(2π√LC)

0016 D9 1E 0000 R          FSTP RESO           ;save frequency

001A 9B                    FWAIT
001B CB                    RET

001C                FREQ   ENDP

001C                CODE   ENDS

                           END
```

Notice the straightforward manner in which the procedure solves this equation. Very little extra data manipulation is required because of the stack inside the coprocessor. Also notice how the constant TWO is defined for the program and how the DIVRP, using classic stack addressing, is used to form the reciprocal. If you own a reverse-polish entry calculator, such as the ones produced by Hewlett-Packard, you are familiar with stack addressing.

## Finding the Roots Using the Quadratic Equation

This example illustrates how to find the roots of a polynomial expression ($ax^2 + bx + c = 0$) using the quadratic equation. The quadratic equation is $(b \pm \sqrt{b^2 - 4ac})/2a$. Example 8–10 illustrates a procedure that finds the roots (R1 and R2) for the quadratic equation. The constants are stored in memory locations A, B, and C. Note that no attempt is made to determine whether the roots are real or imaginary. In practice, you should check the result after the FSQRT instruction to see if an error has occurred (see status-register bit IE in Table 8–2).

**EXAMPLE 8–10**

```
0000                       CODE    SEGMENT USE16 'CODE'
                           ASSUME CS:CODE

0000 40000000              TWO     DD    2.0
0004 40800000              FOUR    DD    4.0
0008 3F800000              A       DD    1.0
000C 00000000              B       DD    0.0
0010 C1100000              C1      DD    -9.0
0014 00000000              R1      DD    ?
0018 00000000              R2      DD    ?

001C                       ROOTS   PROC  FAR

001C 2E: D9 06 0000 R      FLD     TWO
0021 2E: D8 0E 0008 R      FMUL    A              ;form 2a
0026 2E: D9 06 0004 R      FLD     FOUR
```

```
002B 2E: D8 0E 0008 R       FMUL  A
0030 2E: D8 0E 0010 R       FMUL  C1            ;form 4ac
0035 2E: D9 06 000C R       FLD   B
003A 2E: D8 0E 000C R       FMUL  B             ;form b^2
003F DE E1                  FSUBR               ;form b^2 - 4ac
0041 D9 E4                  FTST                ;test b - 4ac for zero
0043 9B DF E0               FSTSW AX            ;copy status register to AX
0046 9E                     SAHF                ;move to flags
0047 74 0E                  JZ    ROOTS1        ;if b - 4ac is zero
0049 D9 FA                  FSQRT               ;find squareroot of b - 4ac
004B 9B DF E0               FSTSW AX
004E A9 0001                TEST  AX,1          ;test for invalid error (negative)
0051 74 04                  JZ    ROOTS1
0053 DE D9                  FCOMPP              ;clear stack
0055 EB 1C                  JMP   ROOTS2        ;end

0057                ROOTS1:

0057 2E: D9 06 000C R       FLD   B
005C D8 E1                  FSUB  ST,ST(1)
005E D8 F2                  FDIV  ST,ST(2)
0060 2E: D9 1E 0014 R       FSTP  R1            ;save root1

0065 2E: D9 06 000C R       FLD   B
006A DE C1                  FADD
006C DE F1                  FDIVR
006E 2E: D9 1E 0018 R       FSTP  R2            ;save root2

0073                ROOTS2:
0073 9B                     FWAIT
0074 CB                     RET

0075                ROOTS   ENDP

0075                CODE    ENDS

                            END   ROOTS
```

## Using a Memory Array to Store Results

The next programming example illustrates the use of a memory array and the scaled-indexed addressing mode to access the array. Example 8–11 shows a procedure that calculates 100 values of inductive reactance. The equation for inductive reactance is $XL = 2\pi FL$. In this example, we use a frequency range from 10Hz to 1000Hz for F and an inductance of 4H. Notice how the instruction FSTP DWORD PTR CS:[EDI+4*ECX] is used to store the reactance for each frequency, beginning with the last at 1000Hz and ending with the first at 10Hz. Also notice how the FCOMP instruction is used to clear the stack just before the RET instruction.

**EXAMPLE 8–11**

```
                    ;a program that calculates the inductive reactance of L
                    ;at a frequency range of 10Hz to 1000Hz and stores them
                    ;in array XL.  Note the increment is 10Hz
                    ;
0000                FINDXL PROC FAR

0000 66| B9 00000064        MOV  ECX,100          ;load count
0006 66| BF 00000035 R      MOV  EDI,OFFSET XL-4  ;address result
```

```
000C D9 EB                    FLDPI                    ;get PI
000E D8 C0                    FADD ST,ST(0)            ;form 2*PI
0010 2E: D8 0E 0035 R         FMU  LL                  ;form 2*PI*L
0015                    L1:
0015 2E: D9 06 01C9 R         FLD  F                   ;get F
001A D8 C9                    FMUL ST,ST(1)
001C 67& 2E: D9 1C 8F         FSTP DWORD PTR [EDI+4*ECX]
0021 2E: D9 06 01C9 R         FLD  F
0026 2E: D8 26 01CD R         FSUB TEN                 ;change frequency
002B 2E: D9 1E 01C9 R         FSTP F
0030 E2 E3                    LOOP L1
0032 D8 D9                    FCOMP
0034 CB                       RET

0035                    FINDXL ENDP

0035 40800000           L      DD   4.0               ;4.0H test value
0039 0064 [             XL     DD   100 DUP (?)
             00000000
                        ]
01C9 447A0000           F      DD   1000.0            ;start at 1000Hz
01CD 41200000           TEN    DD   10.0              ;increment of 10Hz
```

## Displaying a Single-Precision, Floating-Point Number

This section of the text shows how to take the floating-point contents of a 32-bit, single-precision, floating-point number and display it on the video display. The procedure displays the floating-point number as a mixed number, with an integer part and a fractional part separated by a decimal point. In order to simplify the procedure we have placed a limit on the display size of the mixed number so the integer portion is a 32-bit binary number and the fraction is a 24-bit binary number. The procedure will not function properly for larger or smaller numbers.

**EXAMPLE 8–12** (pp. 317–319)

```
.386
.387
0000                    DATAS  SEGMENT

0000 C50B0C00           NUMB   DD   -2224.75
0004 0000               TEMP   DW   ?
0006 0000               WHOLE  DW   ?
0008 00000000           FRACT  DD   ?

000C                    DATAS  ENDS

0000                    CODE   SEGMENT USE16 'CODE'

                               ASSUME  CS:CODE,DS:DATAS

                        ;Main program that displays NUMB
                        ;
0000                    MAIN   PROC  FAR

0000 B8 ---- R                 MOV   AX,DATAS
0003 8E D8                     MOV   DS,AX
0005 E8 0013 R                 CALL  DISP             ;display NUMB
0008 B4 4C                     MOV   AH,4CH           ;exit to DOS
000A CD 21                     INT   21H
```

```
000C                    MAIN    ENDP

000C                    DISPS   PROC    NEAR

000C B4 06                      MOV     AH,6            ;display AL
000E 8A D0                      MOV     DL,AL
0010 CD 21                      INT     21H
0012 C3                         RET

0013                    DISPS   ENDP

0013                    DISP    PROC    NEAR

0013 9B D9 3E 0004 R            FSTCW   TEMP            ;set rounding to chop
0018 81 0E 0004 R 0C00          OR      TEMP,0C00H
001E 9B D9 2E 0004 R            FLDCW   TEMP

0023 D9 06 0000 R               FLD     NUMB            ;get NUMB
0027 D9 E4                      FTST                    ;test NUMB
0029 9B DF E0                   FSTSW   AX              ;status to AX
002C 25 4500                    AND     AX,4500H        ;get C3, C2, and C0
002F 3D 0100                    CMP     AX,0100H        ;test for -
0032 75 05                      JNE     DISP1           ;if positive
0034 B0 2D                      MOV     AL,'-'
0036 E8 000C R                  CALL    DISPS           ;display minus

0039                    DISP1:

0039 D9 E1                      FABS                    ;make ST positive
003B D9 FC                      FRNDINT                 ;get integer
003D DF 16 0006 R               FIST    WHOLE           ;store integer
0041 D9 06 0000 R               FLD     NUMB
0045 D9 E1                      FABS
0047 DE E9                      FSUB                    ;get fraction
0049 D9 E1                      FABS
004B D9 1E 0008 R               FSTP    FRACT           ;save fraction
004F 9B                         FWAIT

                        ;display integer part

0050 A1 0006 R                  MOV     AX,WHOLE
0053 B9 0000                    MOV     CX,0
0056 BB 000A                    MOV     BX,10

0059                    DISP2:

0059 41                         INC     CX
005A 33 D2                      XOR     DX,DX
005C F7 F3                      DIV     BX
005E 83 C2 30                   ADD     DX,'0'          ;convert to ASCII
0061 52                         PUSH    DX
0062 0B C0                      OR      AX,AX
0064 75 F3                      JNE     DISP2           ;if not zero

0066                    DISP3:

0066 58                         POP     AX
0067 E8 000C R                  CALL    DISPS           ;display it
006A E2 FA                      LOOP    DISP3
006C B0 2E                      MOV     AL,'.'          ;display decimal point
006E E8 000C R                  CALL    DISPS

                        ;display fractional part
```

```
0071 A1 0008 R              MOV   AX,WORD PTR FRACT
0074 8B 16 000A R           MOV   DX,WORD PTR FRACT+2
0078 B9 0008                MOV   CX,8

007B                  DISP4:

007B D1 E0                  SHL   AX,1
007D D1 D2                  RCL   DX,1
007F E2 FA                  LOOP  DISP4
0081 81 CA 8000             OR    DX,8000H        ;set implied bit
0085 92                     XCHG  AX,DX
0086 BB 000A                MOV   BX,10

0089                  DISP5:

0089 F7 E3                  MUL   BX
008B 50                     PUSH  AX
008C 92                     XCHG  DX,AX
008D 04 30                  ADD   AL,'0'
008F E8 000C R              CALL  DISPS           ;display digit
0092 58                     POP   AX
0093 0B C0                  OR    AX,AX
0095 75 F2                  JNZ   DISP5
0097 C3                     RET

0098                  DISP  ENDP

0098                  CODE  ENDS

                            END   MAIN
```

Example 8–12 lists the procedure for displaying the contents of memory location NUMB on the video display at the current cursor position. The procedure first tests the sign of the number and displays a space for a positive number and a minus sign for a negative number. After displaying the sign, the number is made positive by the FABS instruction. Next, we divide it into integer and fractional parts and store them at WHOLE and FRACT.

The last part of the procedure displays the whole number part followed by the fractional part. Note that the fractional part may contain a certain value for rounding error. We have not adjusted the number to remove the rounding error that is inherent in floating-point fractional numbers.

## Reading a Mixed Number from the Keyboard

If floating-point arithmetic is used in a program, we must have a method of reading the number from the keyboard and converting it to a floating-point number. The procedure listed in Example 8–13 reads a signed mixed number from the keyboard and converts it to a floating-point number located at the top of the stack inside the coprocessor.

**EXAMPLE 8–13** (pp. 319–321)

```
0000                  DATA  SEGMENT

0000 00               SIGN  DB    ?
0001 0000             TEMP1 DW    ?
0003 41200000         TEN   DD    10.0

0007                  DATA  ENDS
```

```
0000                    CODE    SEGMENT USE16 'CODE'

                                ASSUME CS:CODE,DS:DATA

                        ;procedure that reads a mixed number from the keyboard
                        ;and leaves it at the top of the coprocessor stack.

0000                    READ    PROC  FAR

0000 B8 ---- R                  MOV   AX,DATA         ;address data segment
0003 8E D8                      MOV   DS,AX
0005 9B D9 EE                   FLDZ                  ;clear ST
0008 C6 06 0000 R 00            MOV   SIGN,0          ;clear sign
000D E8 007C                    CALL  GET             ;read a character
0010 3C 2D                      CMP   AL,'-'          ;test for minus
0012 75 07                      JNE   READ1           ;if not minus
0014 C6 06 0000 R FF            MOV   SIGN,0FFH       ;set sign for minus
0019 EB 0F                      JMP   READ3           ;get integer part

001B                    READ1:

001B 3C 2B                      CMP   AL,'+'          ;test for plus
001D 74 0B                      JE    READ3           ;get integer part
001F 3C 30                      CMP   AL,'0'          ;test for number
0021 72 06                      JB    READ2
0023 3C 39                      CMP   AL,'9'
0025 77 02                      JA    READ2
0027 EB 04                      JMP   READ4           ;if a number

0029                    READ2:

0029 CB                         RET

002A                    READ3:

002A E8 005F                    CALL  GET             ;read integer part

002D                    READ4:

002D 3C 2E                      CMP   AL,'.'          ;test for fraction
002F 74 27                      JE    READ7           ;if fraction
0031 3C 30                      CMP   AL,'0'          ;test for number
0033 72 17                      JB    READ5
0035 3C 39                      CMP   AL,'9'
0037 77 13                      JA    READ5
0039 9B D8 0E 0003 R            FMUL  TEN             ;form integer
003E 32 E4                      XOR   AH,AH
0040 2C 30                      SUB   AL,'0'
0042 A3 0001 R                  MOV   TEMP1,AX
0045 9B DE 06 0001 R            FIADD TEMP1
004A EB DE                      JMP   READ3

004C                    READ5:

004C 80 3E 0000 R 00            CMP   SIGN,0          ;adjust sign
0051 75 01                      JNE   READ6
0053 CB                         RET

0054                    READ6:

0054 9B D9 E0                   FCHS
0057 CB                         RET
```

```
0058                    READ7:

0058 9B D9 E8                   FLD1                    ;form fraction
005B 9B D8 36 0003 R            FDIV  TEN

0060                    READ8:

0060 E8 0029                    CALL  GET               ;read character
0063 3C 30                      CMP   AL,'0'            ;test for number
0065 72 20                      JB    READ9
0067 3C 39                      CMP   AL,'9'
0069 77 1C                      JA    READ9
006B 32 E4                      XOR   AH,AH
006D 2C 30                      SUB   AL,'0'
006F A3 0001 R                  MOV   TEMP1,AX
0072 9B DF 06 0001 R            FILD  TEMP1             ;load number
0077 9B D8 C9                   FMUL  ST,ST(1)          ;form fraction
007A 9B DC C2                   FADD  ST(2),ST
007D 9B D8 D9                   FCOMP
0080 9B D8 36 0003 R            FDIV  TEN
0085 EB D9                      JMP   READ8

0087                    READ9:

0087 9B D8 D9                   FCOMP                   ;clear stack
008A EB C0                      JMP   READ5
008C                    READ    ENDP

008C                    GET     PROC  NEAR

008C B4 06                      MOV   AH,6              ;read character
008E B2 FF                      MOV   DL,0FFH
0090 CD 21                      INT   21H
0092 74 F8                      JZ    GET
0094 C3                         RET

0095                    GET     ENDP

0095                    CODE    ENDS

                                END
```

First, the sign is read from the keyboard, if present and saved for later use in adjusting the sign of the resultant floating-point number. Next, the integer portion of the number is read. This portion terminates with a period, space, or carriage return. If a period is typed, the procedure continues and reads the fractional part. If a space or carriage return is entered, the number is converted to floating-point form.

## 8–5 SUMMARY

1. The arithmetic coprocessor functions in parallel with the microprocessor. This means the microprocessor and coprocessor can execute their instructions simultaneously.
2. The data types manipulated by the coprocessor include signed-integer, floating-point, and binary-coded decimal (BCD).

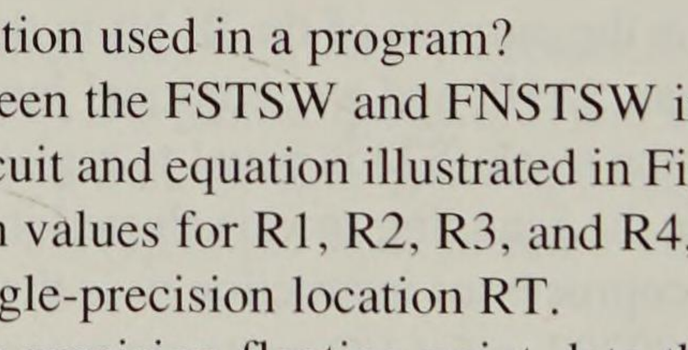
3. There are three forms of integers used with the coprocessor: word (16 bits), short (32 bits) and long (64 bits). Each integer contains a signed number, in true magnitude for positive numbers and two's complement form for negative numbers.
4. A BCD number is stored as an 18-digit number in 10 bytes of memory. The most-significant byte contains the sign bit, and the remaining 9 bytes contain an 18-digit packed BCD number.
5. The coprocessor supports three types of floating-point numbers: single-precision (32 bits), double-precision (64 bits), and extended-precision (80 bits). A floating-point number has three parts: the sign, the biased exponent, and the significand. In the coprocessor, the exponent is biased with a constant. The integer bit of the normalized number is not stored in the significand except in the extended-precision form.
6. Decimal numbers are converted to floating-point numbers by: (a) converting the number to binary, (b) normalizing the binary number, (c) adding the bias to the exponent, and (d) storing the number in floating-point form.
7. Floating-point numbers are converted to decimal by: (a) subtracting the bias from the exponent, (b) unnormalizing the number, and (c) converting it to decimal.
8. The 80287 uses I/O space for the execution of some of its instructions. This space is invisible to the program and is used internally by the 80286/80287 system. The 16-bit I/O addresses 00F8H, 00FAH, and 00FCH must not be used for I/O data transfers in a system that contains an 80287. The 80387, 80486/7, and Pentium use I/O address 80000000H.
9. The coprocessor contains a status register that indicates busy, conditions that follow a compare or test, the location of the top of the stack, and the state of the error bits.
10. The control register of the coprocessor contains control bits that select infinity, rounding, precision, and error masks.
11. The following directives are often used with the coprocessor for storing data: DW (define word), DD (define doubleword), DQ (define quadword) and DT (define 10 bytes).
12. The coprocessor uses a stack to transfer data between itself and the memory system. Generally, data are loaded to the top of the stack or removed from the top of the stack for storage.
13. Internal coprocessor data are always in the 80-bit extended-precision form. The only time data are in any other form is when they are stored or loaded from the memory.
14. The coprocessor addressing modes include the classic stack mode, register, register with a pop, and memory. Stack addressing is implied, the data at ST become the source, ST(1) becomes the destination, and the result is found in ST after a pop.
15. The coprocessor's arithmetic operations include addition, subtraction, multiplication, division, and square root.
16. There are transcendental functions in the coprocessor's instruction set. These functions find the partial tangent or arctangent, $2^x - 1$, $Y \log_2 X$, and $Y \log_2(X + 1)$. The 80387, 80486/7, and Pentium also include sine and cosine functions.
17. Constants stored inside the coprocessor provide +0.0, +1.0, $\pi$, $\log_2 10$, $\log_2 e$, $\log_{10} 2$, and $\log_e 2$.
18. The 80387 functions with the 80386 microprocessor, and the 80487SX functions with the 80486SX microprocessor, but the 80486DX and Pentium contain their own internal arithmetic coprocessors. The instructions performed by the earlier versions are available on these coprocessors.

## 8–6 QUESTIONS AND PROBLEMS

1. List the three types of data that are loaded or stored in memory by the coprocessor.
2. List the three integer data types, the range of the integers stored in them, and the number of bits allotted to each.
3. Explain how a BCD number is stored in memory by the coprocessor.
4. List the three types of floating-point numbers used with the coprocessor and the number of binary bits assigned to each.
5. Convert the following decimal numbers into single-precision, floating-point numbers
   (a) 28.75
   (b) 624
   (c) –0.615
   (d) +0.0
   (e) –1000.5
6. Convert the following single-precision, floating-point numbers into decimal:
   (a) 11000000 11110000 00000000 00000000
   (b) 00111111 00010000 00000000 00000000
   (c) 01000011 10011001 00000000 00000000
   (d) 01000000 00000000 00000000 00000000
   (e) 01000001 00100000 00000000 00000000
   (f) 00000000 00000000 00000000 00000000
7. Explain what the coprocessor does when a normal microprocessor instruction executes.
8. Explain what the microprocessor does when a coprocessor instruction executes.
9. What is the purpose of the C3–C0 bits in the status register?
10. What is the purpose of the IE bit in the status register?
11. How can SAHF and a conditional jump instruction be used to determine whether the top of the stack (ST) is equal to register ST(2)?
12. How is the rounding mode selected in the 80X87?
13. What coprocessor instruction uses the microprocessor's AX register?
14. In the 80287, what I/O ports are reserved for coprocessor use?
15. How are data stored inside the coprocessor?
16. What is a NAN?
17. Whenever the coprocessor is reset, the top of the stack register is register number _____.
18. What does the term *chop* mean in the rounding control bits of the control register?
19. What is the difference between *affine* and *projective infinity* control?
20. What microprocessor instruction forms the opcodes for the coprocessor?
21. The FINIT instruction selects _____-precision for all coprocessor operations.
22. Using assembler pseudo-opcodes, form a statement that accomplishes the following:
    (a) Store a 23.44 into a double-precision, floating-point memory location named FROG.
    (b) Store a –123 into a 32-bit signed integer location named DATA3.
    (c) Store a –23.8 into a single-precision, floating-point memory location named DATA1.
    (d) Reserve a double-precision memory location named DATA2.

23. Describe how the FST DATA instruction functions. Assume that DATA is defined as a 64-bit memory location.
24. What does the FILD DATA instruction accomplish?
25. Form an instruction that adds the contents of register 3 to the top of the stack.
26. Describe the operation of the FADD instruction.
27. Choose an instruction that subtracts the contents of register 2 from the top of the stack and stores the result in register 2.
28. What is the function of the FBSTP DATA instruction?
29. What is the difference between a forward and a reverse division?
30. What is the difference between the FTST instruction and FXAM?
31. Explain what the F2XM1 instruction calculates.
32. Which coprocessor status-register bit should be tested after the FSQRT instruction executes?
33. Which coprocessor instruction pushes $\pi$ onto the top of the stack?
34. Which coprocessor instruction places a 1.0 at the top of the stack?
35. What will FFREE ST(2) accomplish when executed?
36. Which instruction stores the environment?
37. What does the FSAVE instruction save?
38. Develop a procedure that finds the area of a rectangle ($A = L \times W$). Memory locations for this procedure are single-precision, floating-point locations A, L, and W.
39. Write a procedure that finds the capacitive reactance ($XL = 1/2\pi FC$). Memory locations for this procedure are single-precision, floating-point locations XL, F, and C.
40. Develop a procedure that generates a table of square roots for the integers 2 through 10. The results must be stored as single-precision, floating-point numbers in an array called ROOTS.
41. When is the FWAIT instruction used in a program?
42. What is the difference between the FSTSW and FNSTSW instruction?
43. Given the series/parallel circuit and equation illustrated in Figure 8–12, develop a program, using single-precision values for R1, R2, R3, and R4, that finds the total resistance. Store the result at single-precision location RT.
44. Given two arrays of double-precision floating-point data that each contain 100 elements (ARRAY1 and ARRAY2), develop a procedure that finds the product of ARRAY1 times ARRAY2 and stores the double-precision, floating-point result in a third array (ARRAY3).
45. Develop a procedure that takes the single-precision contents of register EBX times $\pi$ and stores the result in register EBX as a single-precision, floating-point number. You must use memory to accomplish this task.

**FIGURE 8–12**




$$RT = R1 + \cfrac{1}{\cfrac{1}{R2} \quad \cfrac{1}{R3} \quad \cfrac{1}{R4}}$$

# CHAPTER 9

# The 80386 and 80486 Microprocessors

## INTRODUCTION

The 80386 microprocessor is a full 32-bit version of the earlier 8086/8088 and 80286 16-bit microprocessors and represents a major advancement in the architecture. Along with this larger word size are many other improvements and additional features. The 80386 microprocessor features multi-tasking, memory management, virtual memory with or without paging, software protection, and a large memory system. All software written for the 8086/8088 and the 80286 are upward compatible to the 80386 microprocessor. The amount of memory addressable by the 80386 is increased from the 1M bytes found in the 8086/8088 and the 16M bytes found in the 80286, to 4G bytes in the 80386. The 80386 can switch between protected mode and real mode without resetting the microprocessor. Switching from protected mode to real mode was a problem on the 80286 microprocessor because it required a hardware reset.

The 80486 microprocessor is an enhanced version of the 80386 microprocessor that executes many of its instructions in one clocking period. The 80486 microprocessor also contains an 8K-byte cache memory and improved 80387 numeric coprocessor. When the 80486 is operated at the same clock frequency as the 80386, it performs about 50 percent faster. Later, in Chapter 14, we shall see that the Pentium, which contains a 16K cache memory, performs at twice the speed of the 80486 microprocessor. The Pentium also contains an improved numeric coprocessor that operates at five times the speed of the 80486 numeric coprocessor.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Contrast the 80386/80486 microprocessor with earlier Intel microprocessors.
2. Describe the organization and interface of the 32-bit 80386/80486 memory system.
3. Describe the operation of the 80386/80486 memory management unit and paging unit.

4. Switch between protected mode and real mode.
5. Define the operation of additional 80386/80486 instructions and addressing modes.
6. Explain the operation of a cache memory system.
7. Detail the interrupt structure and direct memory access structure of the 80386/80486.
8. Contrast the 80486 with the 80386 microprocessor.
9. Explain the operation of the 80486 cache memory.

## 9–1 INTRODUCTION TO THE 80386 MICROPROCESSOR

Before this or any other microprocessor can be used in a system, the function of each pin must be understood. This section of the chapter details the operation of each pin along with the external memory system and I/O structures of the 80386.

Figure 9–1 illustrates the pinout of the 80386DX microprocessor, which is packaged in a 132-pin PGA (pin grid array). Two versions of the 80386 are commonly available. The 80386DX, illustrated and described in this chapter, is the full version. The 80386SX is a reduced bus version of the 80386DX. A new version of the 80386—the 80386EX—incorporates the AT bus system, dynamic RAM controller, programmable chip selection logic, 26 address pins, 16 data pins, and 24 I/O pins. The 80386DX addresses 4G bytes of memory through its 32-bit data bus and 32-bit address. The 80386SX, more like the 80286, addresses 16M bytes of memory with its 24-bit address bus via its 16-bit data bus. The 80386SX was developed after the 80386DX for applications that didn't require the full 32-bit bus version. The 80386SX is found in many personal computers that use the same basic

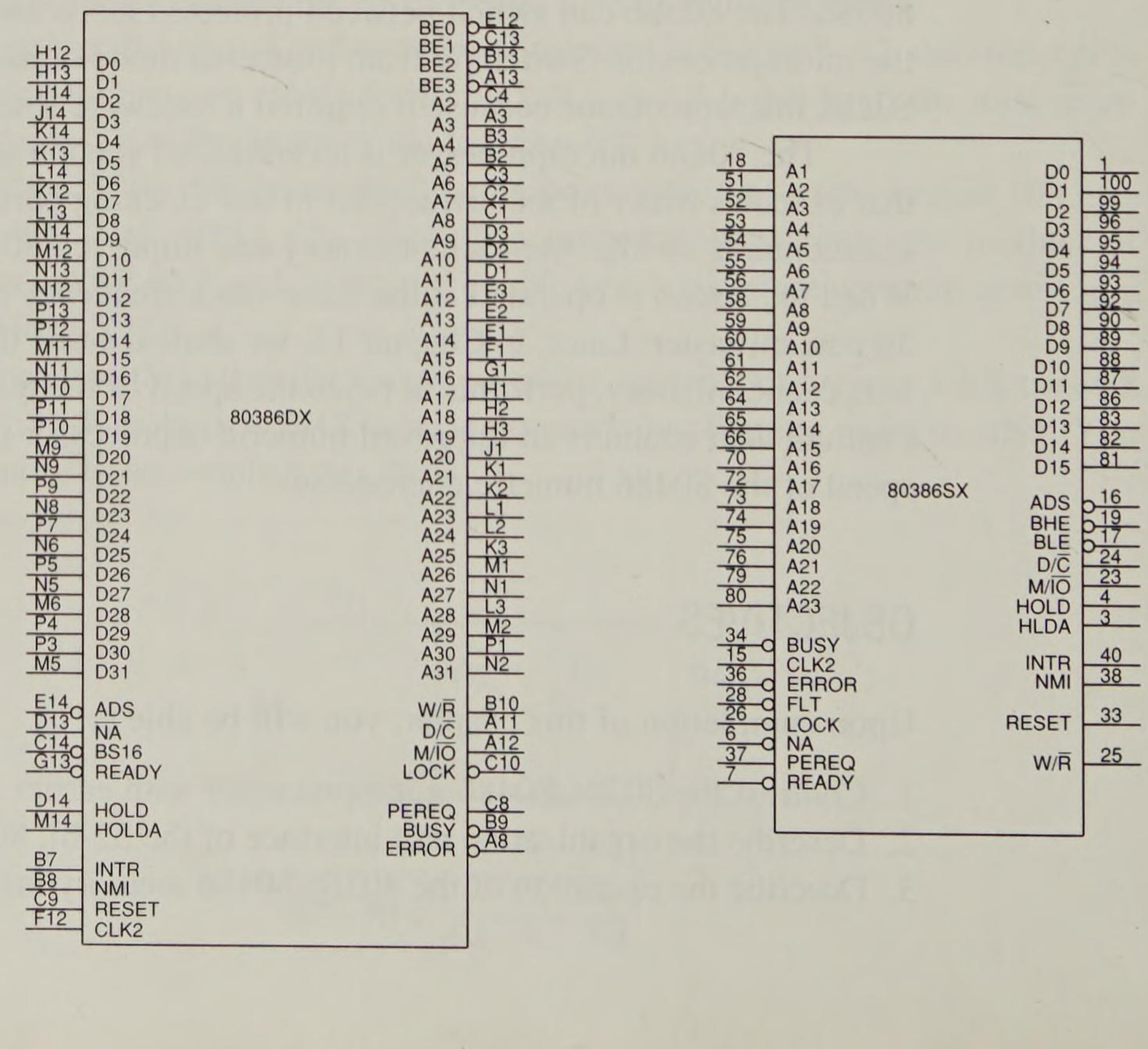



**FIGURE 9–1** The pinouts of the 80386DX and 80386SX microprocessors.

motherboard design as the 80286. At this time most applications, including Windows, require less than 16M bytes of memory, so the 80386SX is a fairly popular and less costly version of the 80386 microprocessor. The 80486SX is becomming a less expensive upgrade path for newer systems, but the 80386 can still be used for many applications.

As with earlier versions of the Intel family of microprocessors, the 80386 requires a single +5.0 V power supply for operation. The power supply current averages 550 mA for the 25 MHz version of the 80386, 500 mA for the 20 MHz version, and 450 mA for the 16 MHz version. Also available is a 33 MHz version that requires 600 mA of power supply current. Note that during some modes of normal operation, power supply current can surge to over 1.0 A. This means that the power supply and power distribution network must be capable of supplying these current surges. This device contains multiple Vcc and Vss connections that must all be connected to +5.0 V and ground for proper operation. Some of the pins are labeled N/C (no connection) and must not be connected. Additional versions of the 80386SX are available with a +3.3 V power supply. These are often found in portable notebook or laptop computers and are usually packaged in a surface mount device.

Each 80386 output pin is capable of providing 4.0 mA (address and data connections) or 5.0 mA (other connections). This represents an increase in drive current compared to the 2.0 mA available on earlier 8086, 8088, and 80286 output pins. Each input pin represents a small load requiring only ±10 μA of current. In some systems, except the smallest, these current levels require bus buffers. Table 9–1 shows the input current requirements of some popular logic family devices that are often connected to the microprocessor output pins. This table allows fanouts (the number of gates connected to an output) to be determined for various logic families. For example, if 74LSXXX gates are connnected to an address connection on the 80386, we can attach up to 10 such gates, because the 80386 provides up to 4.0 mA, and the 74LSXXX requires only 0.4 mA of current per input.

The function of each 80386DX group of pins follows:

A31–A2 (Address Bus Connections)—address any of the 1G × 32 memory locations found in the 80386 memory system. Note that A0 and A1 are encoded in the bus enable ($\overline{BE3}$–$\overline{BE0}$), described elsewhere, to select any or all of the four bytes in a 32-bit wide memory location. Also note that because the 80386SX contains a 16-bit data bus in place of the 32-bit data bus found on the

**TABLE 9–1** Logic family input current requirements

| *Family* | *Sink Current* | *Source Current* |
|---|---|---|
| TTL (74XXX) | –1.6 mA | 40 μA |
| TTL (74LSXXX) | –0.4 mA | 20 μA |
| TTL (74SXXX) | –2.0 mA | 50 μA |
| TTL (74ALSXXX) | –0.1 mA | 20 μA |
| TTL (74ASXXX) | –0.5 mA | 25 μA |
| TTL (74FXXX) | –0.5 mA | 25 μA |
| CMOS (74HCXXX) | –1.0 μA | 1.0 μA |
| CMOS (CD4XXX) | –1.0 μA | 1.0 μA |
| NMOS | –10 μA | 10 μA |

80386DX, A1 is present on the 80386SX and the bank selection signals are replaced with $\overline{BHE}$ and $\overline{BLE}$. The $\overline{BHE}$ signal enables the upper data bus half, and the $\overline{BLE}$ signal enables the lower data bus half.

D31–D0 (Data Bus Connections)—transfer data between the microprocessor and its memory and I/O system. Note that the 80386SX contains D15–D0.

$\overline{BE3}$–$\overline{BE0}$ (Bank Enable Signals)—select the access of a byte, word, or doubleword of data. These signals are generated internally by the microprocessor from address bits A1 and A0. On the 80386SX, these pins are replaced by $\overline{BHE}$, $\overline{BLE}$, and A1.

M/$\overline{IO}$ (Memory/IO)—selects a memory device with a logic 1 or an I/O device with a logic 0. During the I/O operation the address bus contains a 16-bit I/O address on address connections A15–A2.

W/$\overline{R}$ (Write/Read)—indicates that the current bus cycle is a write with a logic 1 or a read with a logic 0.

$\overline{ADS}$ (Address Data Strobe)—becomes active whenever the 80386 has issued a valid memory or I/O address. This signal is combined with the W/$\overline{R}$ signal to generate the separate read and write signals present in the earlier 8086–80286 microprocessor-based systems.

RESET (Reset)—initializes the 80386, causing it to begin executing software at memory location FFFFFFF0H. The 80386 is reset to the real mode and the left-most 12 address connections remain logic 1s (FFFH) until a far jump or far call is executed. This allows compatibility with earlier microprocessors.

CLK2 (Clock Times 2)—is driven by a clock signal that is twice the operating frequency of the 80386. For example, to operate the 80386 at 16 MHz, we apply a 32 MHz clock to this pin.

$\overline{READY}$ (Ready)—controls the number of wait states inserted into the timing to lengthen memory accesses.

$\overline{LOCK}$ (Lock)—becomes a logic 0 whenever an instruction is prefixed with the LOCK: prefix. This is most often used during DMA accesses.

D/$\overline{C}$ (Data/Control)—indicates the data bus contains data for or from memory or I/O when a logic 1. If D/$\overline{C}$ is a logic 0, the microprocessor is either halted or executing an interrupt acknowledge.

$\overline{BS16}$ (Bus Size 16 Bits)—selects either a 32-bit data bus ($\overline{BS16}$ = 1) or a 16-bit data bus ($\overline{BS16}$ = 0). In most cases, if an 80386DX is operated on a 16-bit data bus, we use the 80386SX that has a 16-bit data bus.

$\overline{NA}$ (Next Address)—causes the 80386 to output the address of the next instruction or data in the current bus cycle. This pin is often used for pipelining the address.

HOLD (Hold)—requests a DMA action.

HLDA (Hold Acknowledge)—indicates that the 80386 is currently in a hold condition.

$\overline{PEREQ}$ (Coprocessor Request)—asks the 80386 to relinquish control and is a direct connection to the 80387 arithmetic coprocessor.

$\overline{\text{BUSY}}$ (Busy)—an input used by the WAIT or FWAIT instruction that waits for the coprocessor to become not busy. This is also a direct connection to the 80387 from the 80386.

$\overline{\text{ERROR}}$ (Error)—indicates to the microprocessor that an error is detected by the coprocessor.

INTR (Interrupt Request)—is used by external circuitry to request an interrupt.

NMI (Nonmaskable Interrupt)—requests a nonmaskable interrupt, as it did on the earlier versions of the microprocessor.

## The Memory System

The physical memory system of the 80386DX is 4G bytes in size and is addressed as such. If virtual addressing is used, 64T bytes are mapped into the 4G bytes of physical space by the memory management unit and descriptors. Note that virtual addressing allows a program to be larger than 4G bytes if a method of swapping with a very large hard-drisk drive exists. Figure 9–2 shows the organization of the 80386DX physical memory system.

The memory is divided into four 8-bit wide memory banks, each containing up to 1G bytes of memory. This 32-bit wide memory organization allows bytes, words, or doublewords of memory data to be accessed directly. The 80386DX transfers up to a 32-bit wide number in a single memory cycle, where the early 8088 requires 4 cycles to accomplish the same transfer, and the 80286 and 80386SX require 2 cycles. Today, data width is important, especially with single-precision, floating-point numbers that are 32 bits in

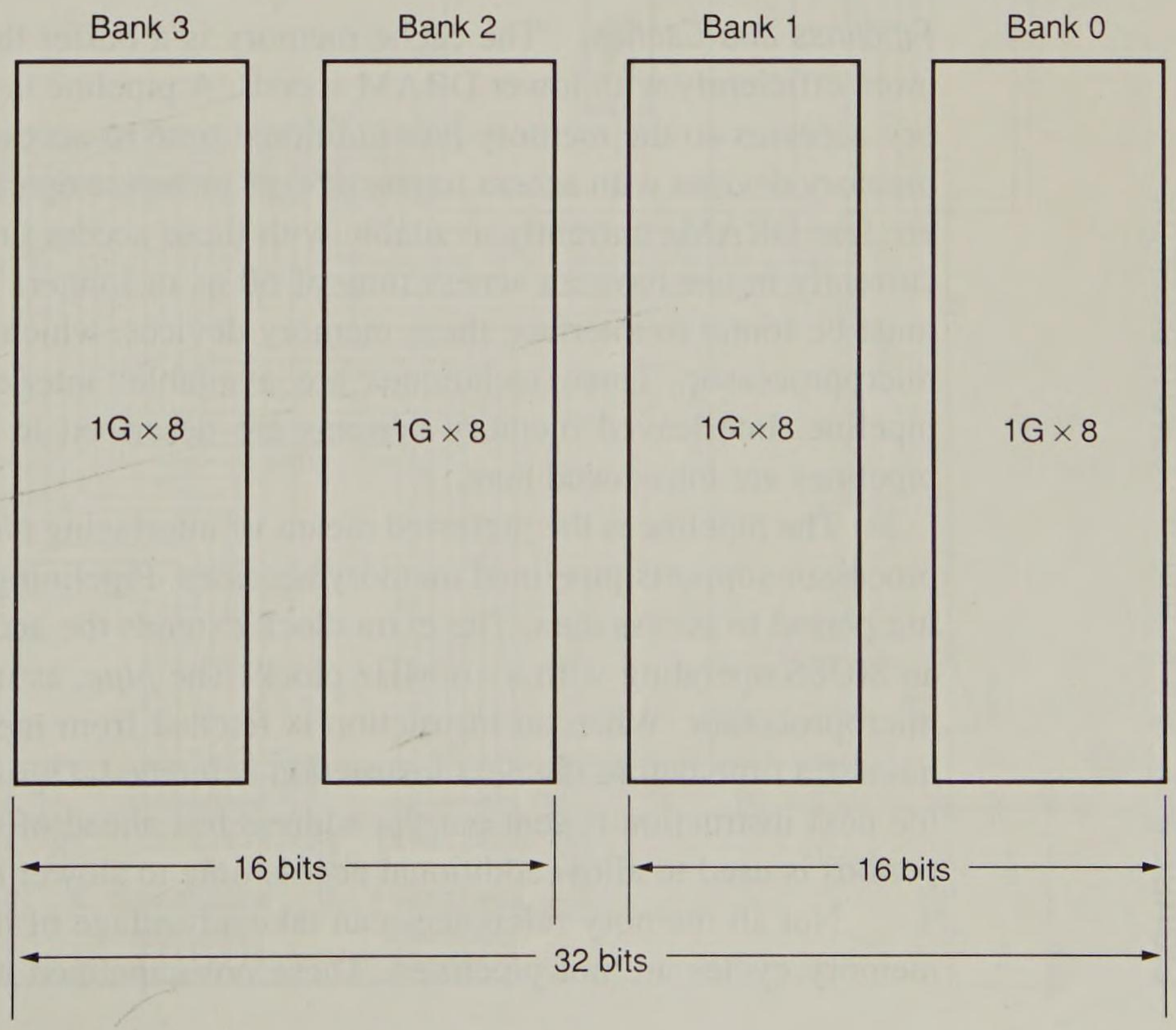



**FIGURE 9–2** The memory system for the 80386 microprocessor. Notice that the memory is organized as four banks, each containing 1G bytes. Memory is accessed as 8-, 16-, or 32-bit data.

width. High-level software normally uses floating-point numbers for data storage, so 32-bit memory locations speed the execution of high-level software written to take advantage of this wider memory.

Memory bytes are numbered in hexadecimal as they were in prior versions of the family. The difference is that the 80386DX uses a 32-bit wide memory address with memory bytes numbered from location 00000000H through FFFFFFFFH.

The two memory banks in the 8086, 80286, and 80386SX systems are accessed via $\overline{\text{BLE}}$ (A0 on the 8086 and 80286) and $\overline{\text{BHE}}$. In the 80386DX, the memory banks are accessed via four bank enable signals $\overline{\text{BE0}}$–$\overline{\text{BE3}}$. This arrangement allows a single byte to be accessed when one bank enable signal is activated by the microprocessor. It also allows a word to be addressed when two bank enable signals are activated. In most cases, a word is addressed in banks 0 and 1 or in banks 2 and 3. Memory location 00000000H is in bank 0, memory location 00000001H is in bank 1, memory location 00000002H is in bank 2, and memory location 00000003H is in bank 3. The 80386DX does not contain address connections A0 and A1, because these have been encoded as the bank enable signals. Likewise the 80386SX does not contain the A0 address pin, because it is encoded in the $\overline{\text{BLE}}$ and $\overline{\text{BHE}}$ signals.

*Buffered System.* Figure 9–3 shows the 80386DX connected to buffers that increase fanout from its address, data, and control connections. This microprocessor is operated at 25 MHz using a 50 MHz clock input signal generated by a integrated oscillator module. Oscillator modules are almost always used to provide a clock in modern microprocessor-based equipment. The HLDA signal is used to enable all buffers in a system that uses direct memory access. Otherwise, the buffer enable pins are connected to a ground in a non-DMA system.

*Pipelines and Caches.* The cache memory is a buffer that allows the 80386 to function more efficiently with lower DRAM speeds. A pipeline is a special way of handling memory accesses so the memory has additional time to access data. A 16 MHz 80386 allows memory devices with access times of 50 ns or less to operate at full speed. Obviously there are few DRAMs currently available with these access times. In fact, the fastest DRAMs currently in use have an access time of 60 ns or longer. This means that some technique must be found to interface these memory devices, which are slower than required by the microprocessor. Three techniques are available: interleaved memory, caching, and a pipeline. Interleaved memory systems are described in Chapter 10, while caching and pipelines are introduced here.

The pipeline is the preferred means of interfacing memory because the 80386 microprocessor supports pipelined memory accesses. Pipelining allows memory an extra clocking period to access data. The extra clock extends the access time from 50 ns to 81 ns on an 80386 operating with a 16 MHz clock. The *pipe*, as it is often called, is set up by the microprocessor. When an instruction is fetched from memory, the microprocessor often has extra time before the next instruction is fetched. During this extra time, the address of the next instruction is sent out the address bus ahead of time. This extra time (one clock period) is used to allow additional access time to slower memory components.

Not all memory references can take advantage of the pipe, which means that some memory cycles are not pipelined. These non-pipelined memory cycles request one wait

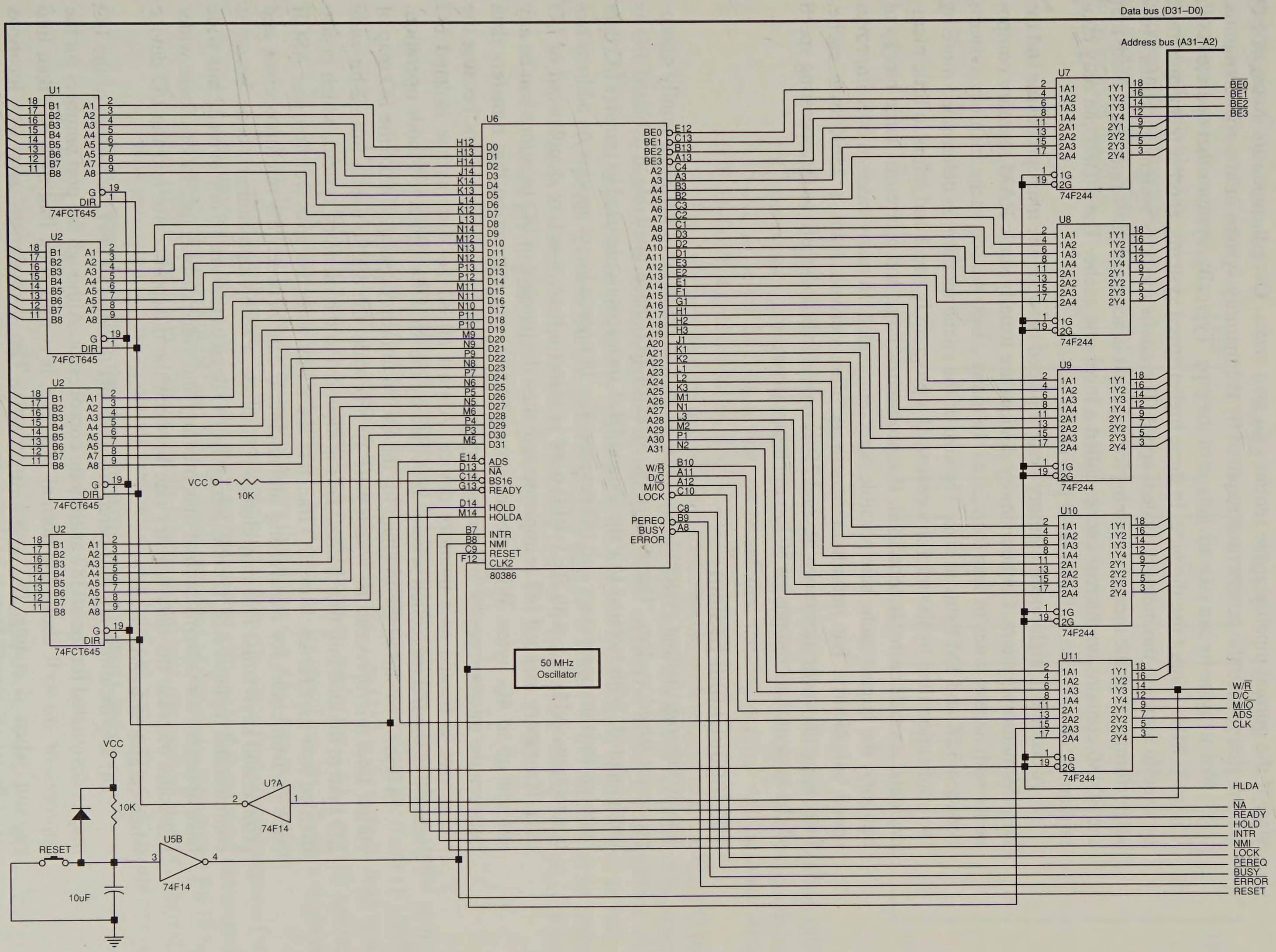



**FIGURE 9–3** A fully buffered 25MHz 80386DX.

state if the normal pipeline cycle requires no wait states. Overall, pipes are a cost-saving feature that reduce the access time required by the memory system in low-speed systems.

Not all systems can take advantage of the pipe. Typically, systems that operate at 20, 25, or 33 MHz cannot use the pipe. In these higher speed systems, another technique must be used to increase the memory system speed. The *cache* memory system improves overall performance of the memory systems for data that is accessed more than once. Note that the 80486 contains an internal cache, called a level one cache. The 80386 can only contain an external cache, called a level two cache.

A cache is a high-speed memory system placed between the microprocessor and the DRAM memory system. Cache memory devices are usually static RAM memory components with access times of less than 25 ns. In many cases, we see cache memory systems between 32K and 1M bytes in size. The size of the cache memory is determined more by the application than by the microprocessor. If a program is small and refers to little memory data, a small cache is beneficial. If a program is large and references large blocks of memory, the largest cache size possible is recommended. In many cases, a 64K improves speed sufficiently, but maximum benefit is often derived from a 256K byte cache. Increasing the cache size much beyond 256K provides little benefit to the operating speed of the system that contains an 80386 microprocessor.

## The Input/Output System

The 80386 input/output system is the same as that found in any Intel 8086 family microprocessor-based system. There are 64K bytes of I/O space available if isolated I/O is implemented. With isolated I/O, the IN and OUT instructions are used to transfer I/O data between the microprocessor and I/O devices. The I/O port address appears on address bus connections A15–A2, with $\overline{BE3}$–$\overline{BE0}$ used to select a byte, word, or doubleword of I/O data. If memory mapped I/O is implemented, then the number of I/O locations can be any amount up to 4G bytes. With memory mapped I/O, any instruction that transfers data between the microprocessor and memory system can be used for I/O transfers, because the I/O device is treated as a memory device. Almost all 80386 systems use isolated I/O because of the I/O protection scheme provided by the 80386 in protected-mode operation.

Figure 9–4 shows the I/O map for the 80386 microprocessor. Unlike the I/O map of earlier Intel microprocessors, which were 16 bits wide, the 80386 uses a full 32-bit wide I/O system divided into four banks. This is identical to the memory system, which is also divided into four banks. Most I/O transfers are 8 bits wide, because we often use ASCII code (a 7-bit code) for transferring alphanumeric data between the microprocessor and printers and keyboards. This may change if Unicode, a 16-bit alphanumeric code, becomes common and replaces ASCII code. Recently, I/O devices that are 16 and even 32 bits wide have appeared for systems such as disk memory and video display interfaces. These wider I/O paths increase the data transfer rate between the microprocessor and the I/O device when compared to 8-bit transfers.

The I/O locations are numbered from 0000H through FFFFH. A portion of the I/O map is designated for the 80387 arithmetic coprocessor. Although the port numbers for the coprocessor are well above the normal I/O map, it is important that they be taken into account when decoding I/O space (overlaps). The coprocessor uses I/O locations 800000F8H–800000FFH for communications between the 80387 and 80386. The 80287

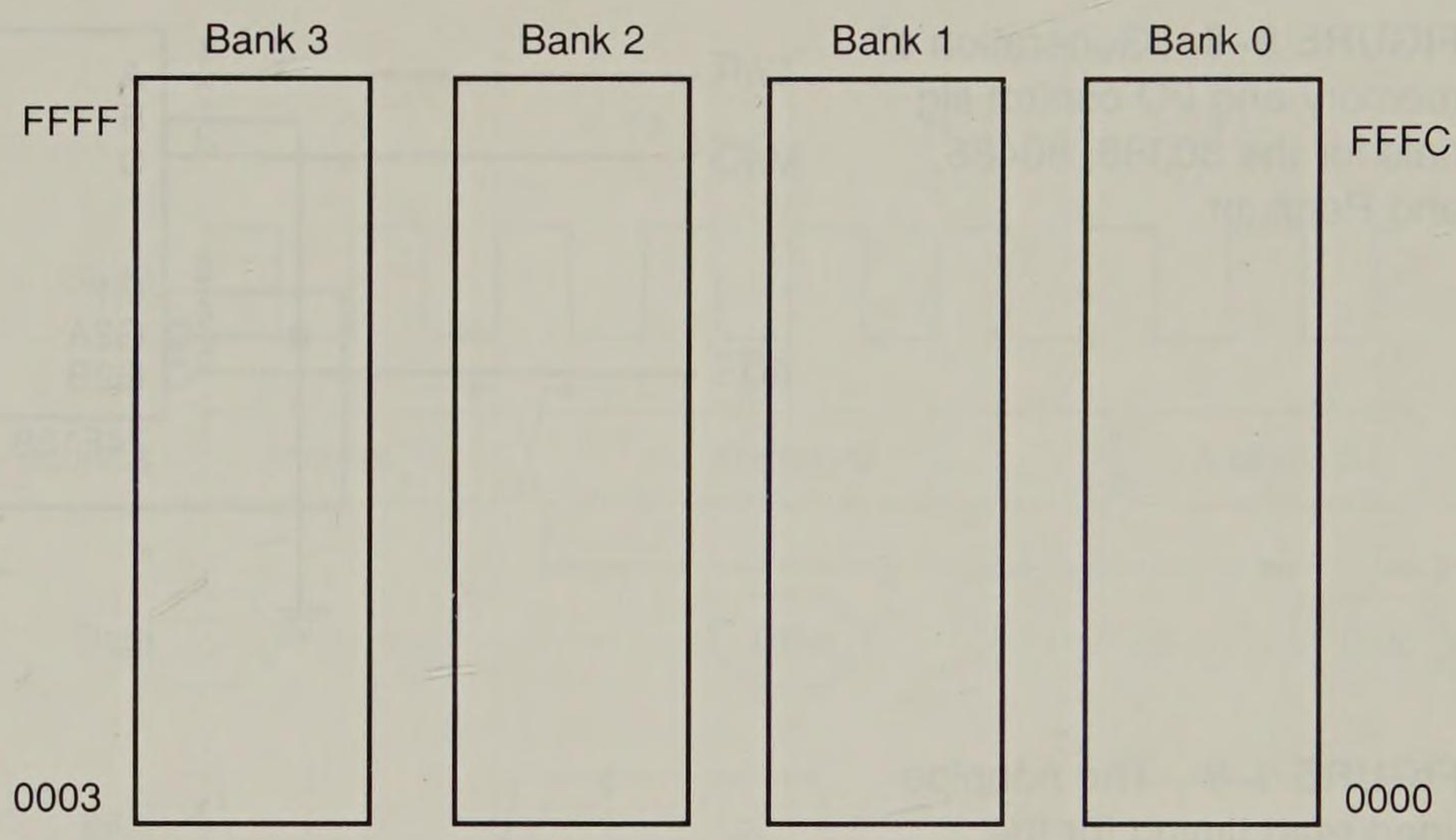



**FIGURE 9–4** The isolated I/O map for the 80386 microprocessor. Here four banks of 8 bits each are used to address 64K different I/O locations. I/O is numbered from location 0000H to FFFFH.

numeric coprocessor designed for use with the 80286 uses I/O addresses 00F8H–00FFH for coprocessor communications. Because we often only decode address connections A15–A2 to select an I/O device, be aware that the coprocessor will activate devices 00F8H–00FFH unless address line A31 is also decoded.

The only new feature that is added to the 80386 with respect to I/O is the I/O privilege information added to the tail-end of the TSS when the 80386 is operated in protected mode. In the protected mode, as described in the section on memory management, an I/O location can be blocked or inhibited. If the blocked I/O location is addressed, an interrupt (type 13, general fault) is generated. This scheme is added so I/O access can be prohibited in a multi-user environment. Blocking is an extension of the protected-mode operation, as are privilege levels.

## Memory And I/O Control Signals

The memory and I/O are controlled with separate signals. The M/$\overline{IO}$ signal indicates whether the data transfer is between the microprocessor and either the memory (M/$\overline{IO}$ = 1) or I/O (M/$\overline{IO}$ = 0). In addition to M/$\overline{IO}$, the memory and I/O systems must read or write data. The W/$\overline{R}$ signal is a logic 0 for a read operation, and a logic 1 for a write operation. The $\overline{ADS}$ signal is used to qualify the M/$\overline{IO}$ and W/$\overline{R}$ control signals. This is a slight deviation from earlier Intel microprocessors that didn't use $\overline{ADS}$ for qualification.

Refer to Figure 9–5 for a simple circuit that generates four control signals for the memory and I/O devices in the system. Notice that two control signals are developed for memory control ($\overline{MRDC}$ and $\overline{MWTC}$) and two control signals are developed for I/O control ($\overline{IORC}$ and $\overline{IOWC}$). These signals are consistent with the memory and I/O control signals generated for use in earlier versions of the Intel microprocessor. They are also used in subsequent chapters that detail memory and I/O interfacing.

## Timing

Timing is important to understanding how to interface memory and I/O to the 80386 microprocessor. Figure 9–6 shows the timing diagram of a nonpipelined memory read

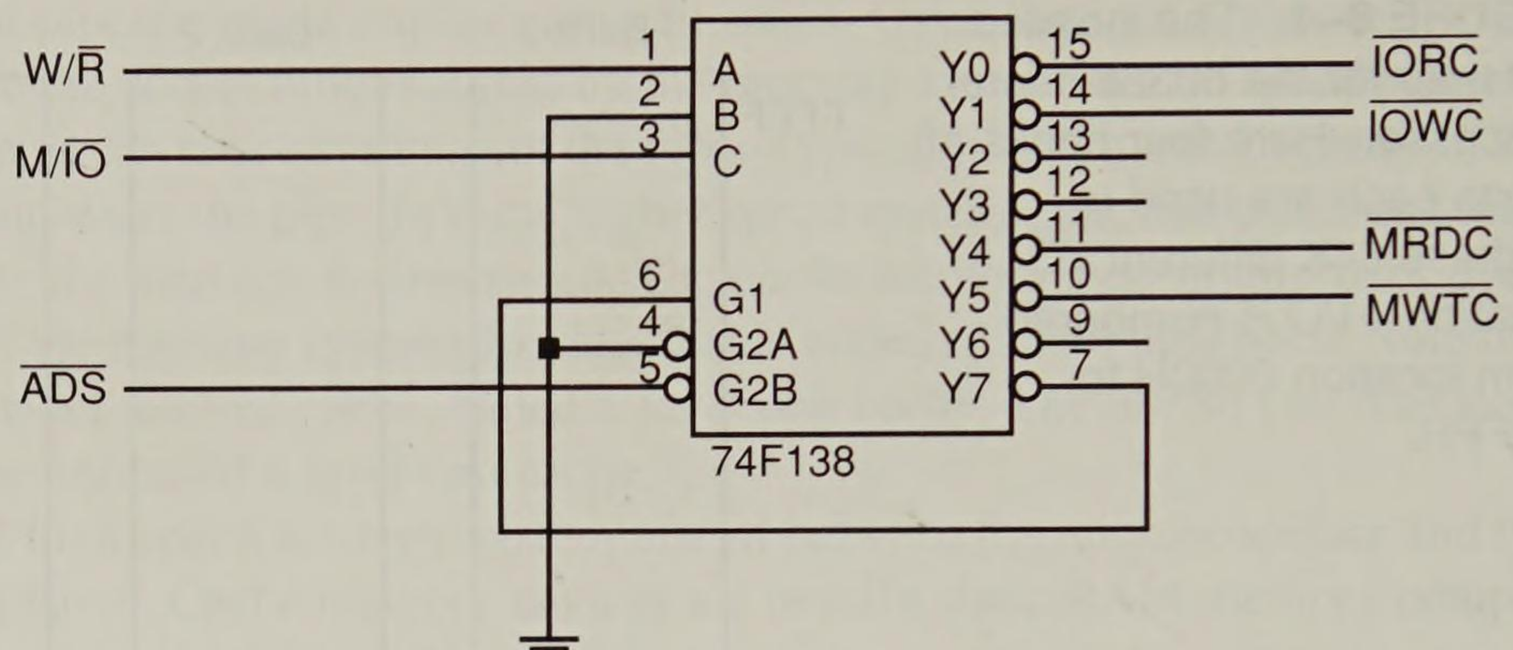



**FIGURE 9–5** Generation of memory and I/O control signals for the 80386, 80486, and Pentium.

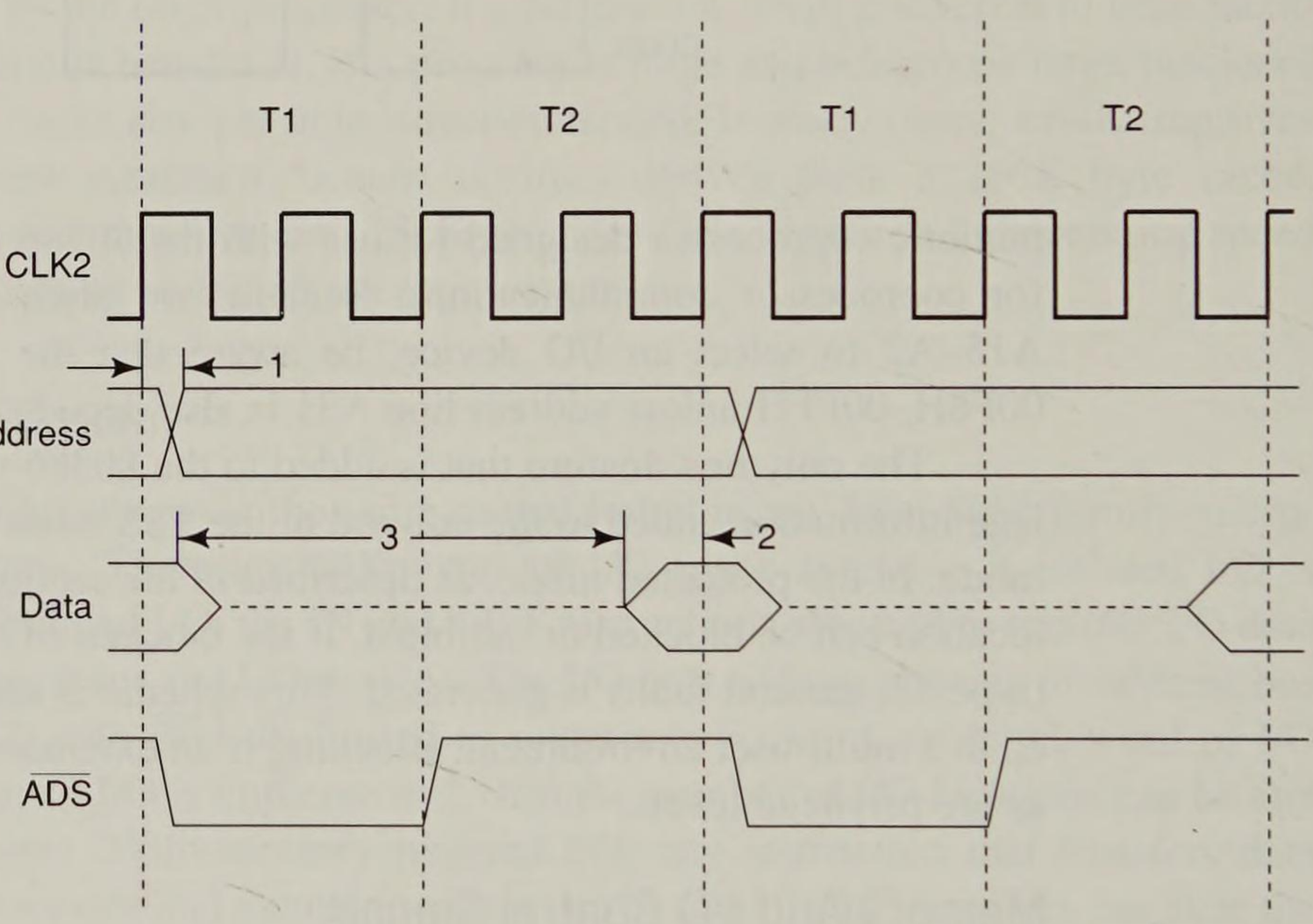



**FIGURE 9–6** The nonpipelined read timing for the 80386 microprocessor.

| | 33 MHz | 25 MHz | 20 MHz | 16 MHz |
|---|---|---|---|---|
| Time 1: | 4–15 ns | 4–21 ns | 4–30 ns | 4–36 ns |
| Time 2: | 5 ns | 7 ns | 11 ns | 11 ns |
| Time 3: | 46 ns | 52 ns | 59 ns | 78 ns |

cycle. Notice that the timing is referenced to the CLK2 input signal, and that a bus cycle consists of four clocking periods.

Each bus cycle contains 2 clocking states; each clocking state (T1 and T2) contains 2 clocking periods. Notice in Figure 9–6 that the access time is listed as time number 3. The 16 MHz version allows memory an access time of 78 ns before wait states are inserted in this nonpipelined mode of operation. To select the nonpipelined mode, we place a logic 1 on the $\overline{NA}$ pin.

Figure 9–7 illustrates the read timing when the 80386 is operated in the pipelined mode. Notice that additional time is allowed to the memory for accessing data because the address is sent out early. Pipelined mode is selected by placing a logic 0 on the $\overline{NA}$ pin

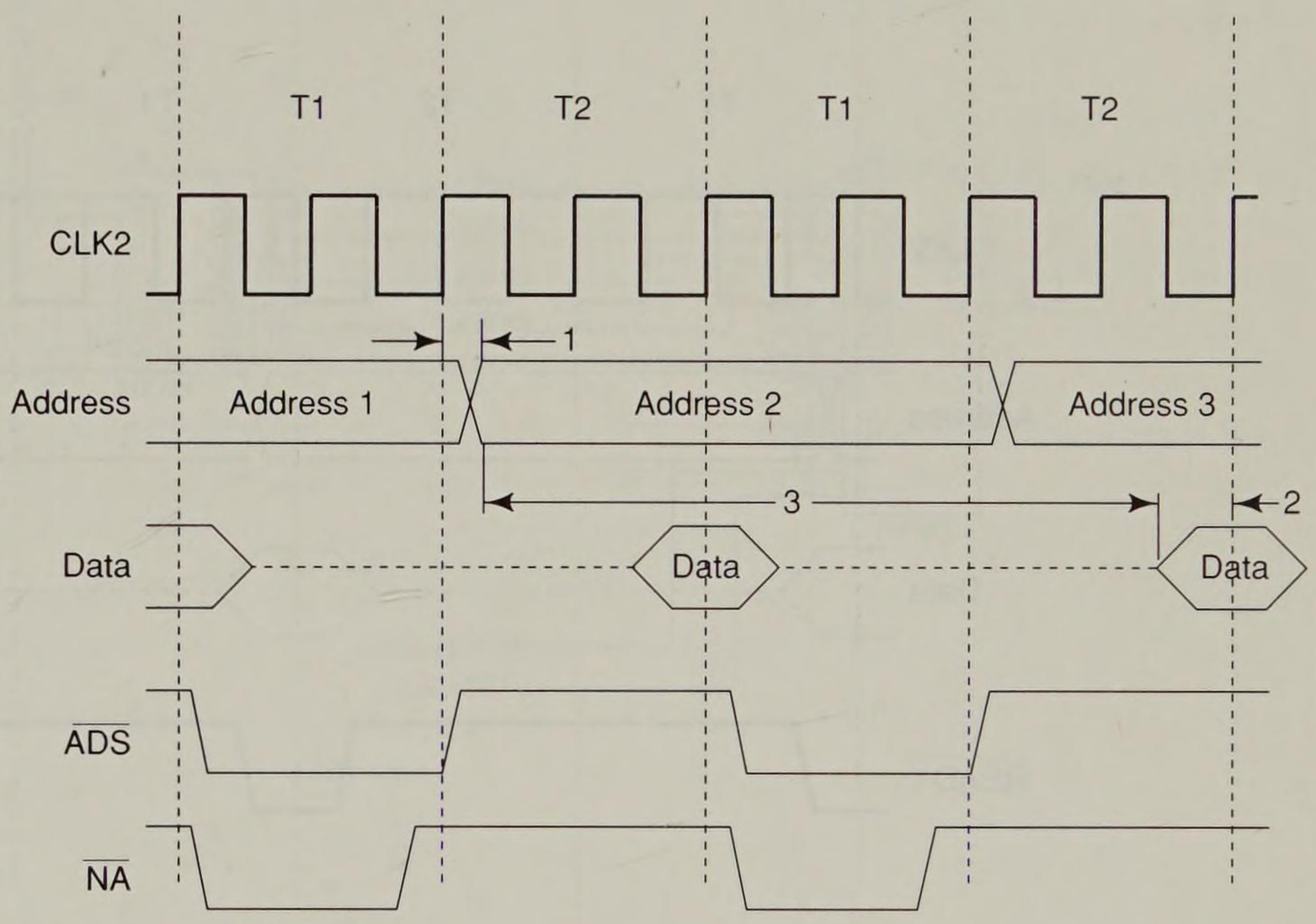



**FIGURE 9–7** The pipelined read timing for the 80386 microprocessor.

| | 33 MHz | 25 MHz | 20 MHz | 16 MHz |
|---|---|---|---|---|
| Time 1: | 4–15 ns | 4–21 ns | 4–30 ns | 4–36 ns |
| Time 2: | 5 ns | 7 ns | 11 ns | 11 ns |
| Time 3: | 80 ns | 92 ns | 109 ns | 145.5 ns |

and by using address latches to capture the pipelined address. The clock pulse that is applied to the address latches comes from the $\overline{\text{ADS}}$ signal. Address latches, as well as interleaved memory banks, must be used with a pipelined system. The minimum number of interleaved banks, two and four, have been used successfully in some applications. Refer to Chapter 10 for additional information on interleaved memory systems.

Notice that the pipelined address appears one complete clocking state before a non-pipelined address would. In the 16 MHz version of the 80386 this allows an additional 67.5 ns for memory access. In a nonpipelined system, memory access time of 78 ns is allowed to the memory system; in a pipelined system, memory access time of 145.5 ns is allowed to the memory system. The advantages of the pipelined system are that, in most bus cycles, no wait states are required, and much lower-speed memory devices may be connected to the microprocessor. The disadvantage of the pipelined system is that we need to interleave memory to use a pipe, which requires additional circuitry and occasional wait states.

## Wait States

Wait states are needed if memory access times are long compared with the time allowed by the 80386 for memory access. In a nonpipelined 33 MHz system, memory access time is only 46 ns. Currently no DRAM memory exists that has an access time of 46 ns. This

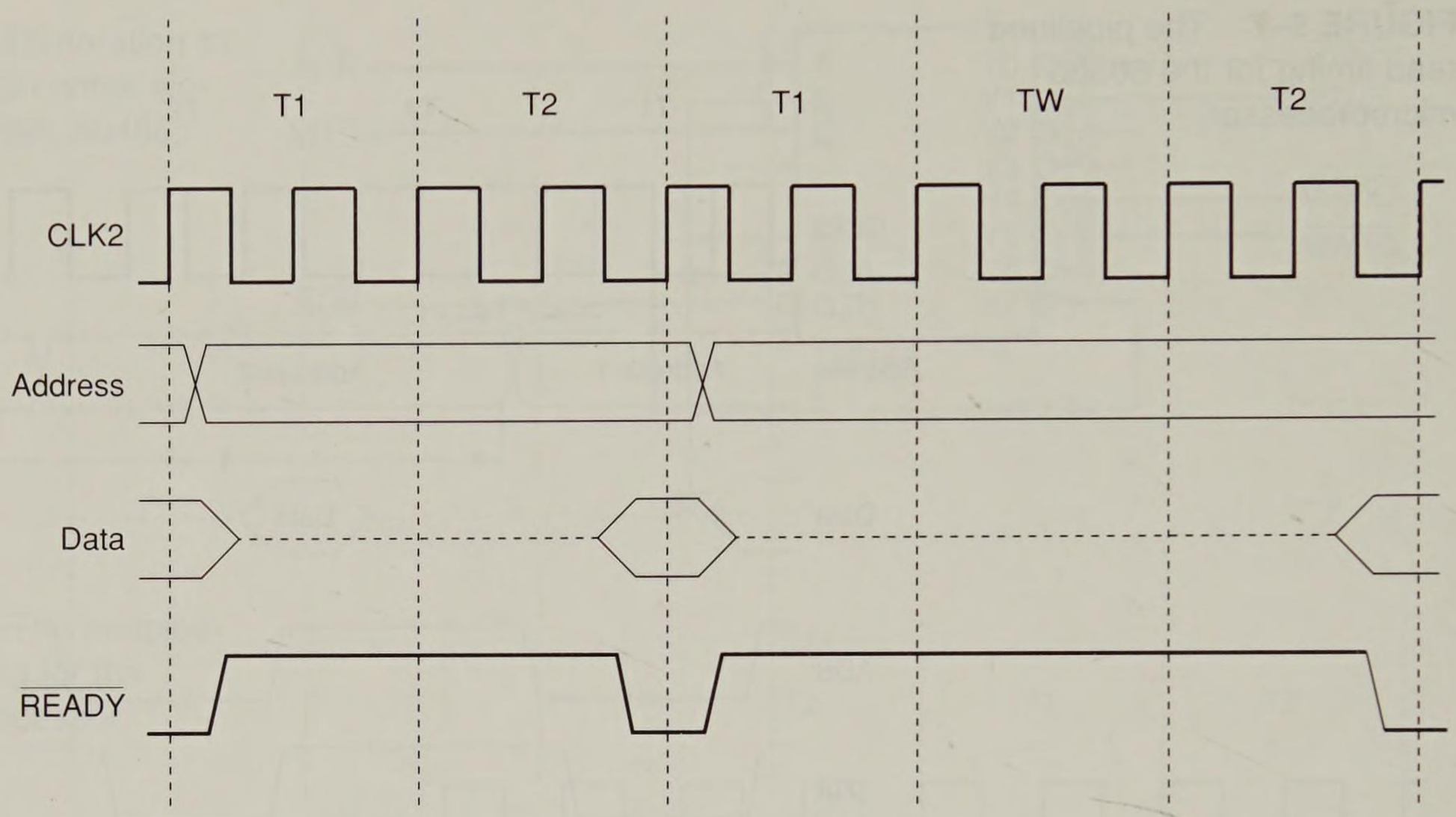



**FIGURE 9–8** A nonpipelined 80386 with 0 and 1 wait states.

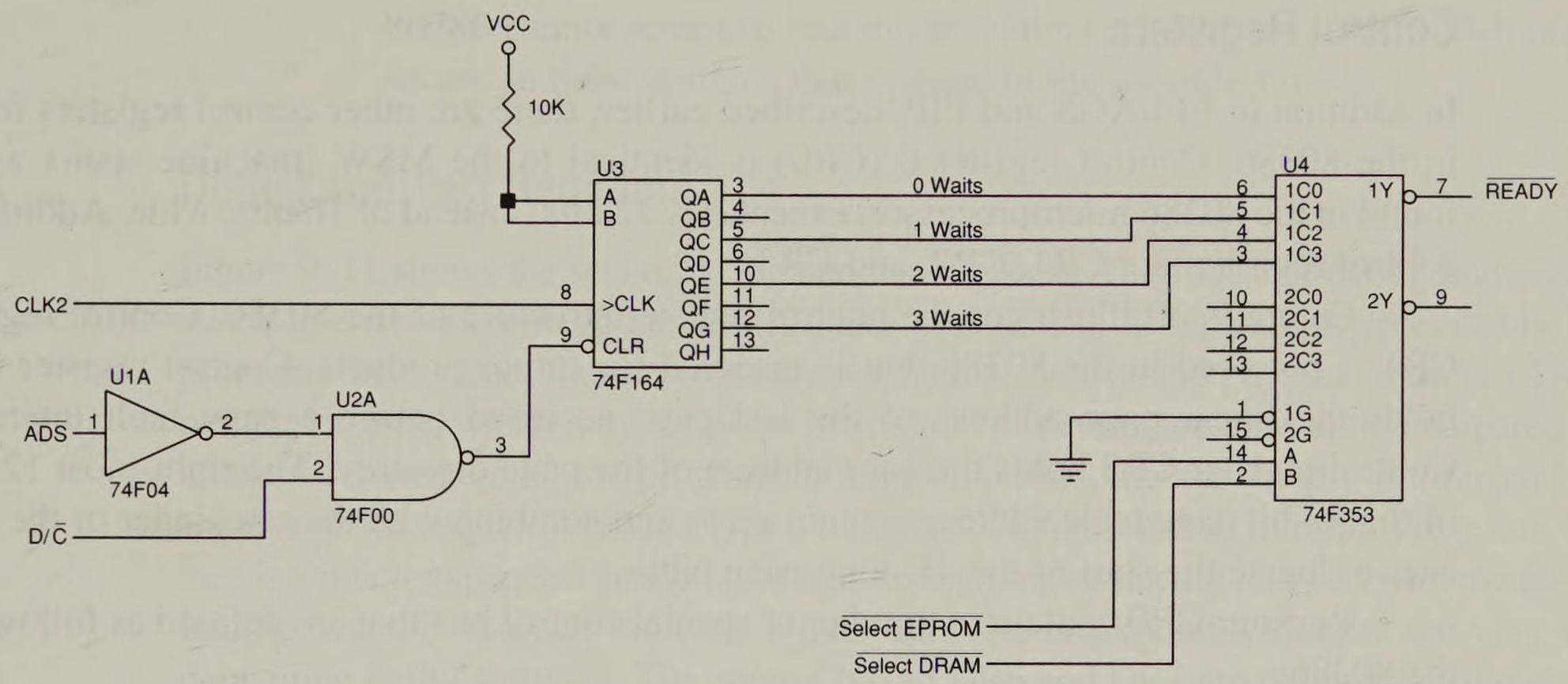
means that wait states must be introduced to increase access time to DRAM (1 wait for 70 ns DRAM) or EPROM (2 waits for 100 ns).

The $\overline{\text{READY}}$ input controls whether or not wait states are inserted into the timing. The $\overline{\text{READY}}$ input on the 80386 is a dynamic input that must be activated during each bus cycle. Figure 9–8 shows a few bus cycles with one 0 wait (normal) cycle and one bus cycle with a 1 wait cycle that contains a single wait state. $\overline{\text{READY}}$ is controlled to cause 0 or 1 wait.

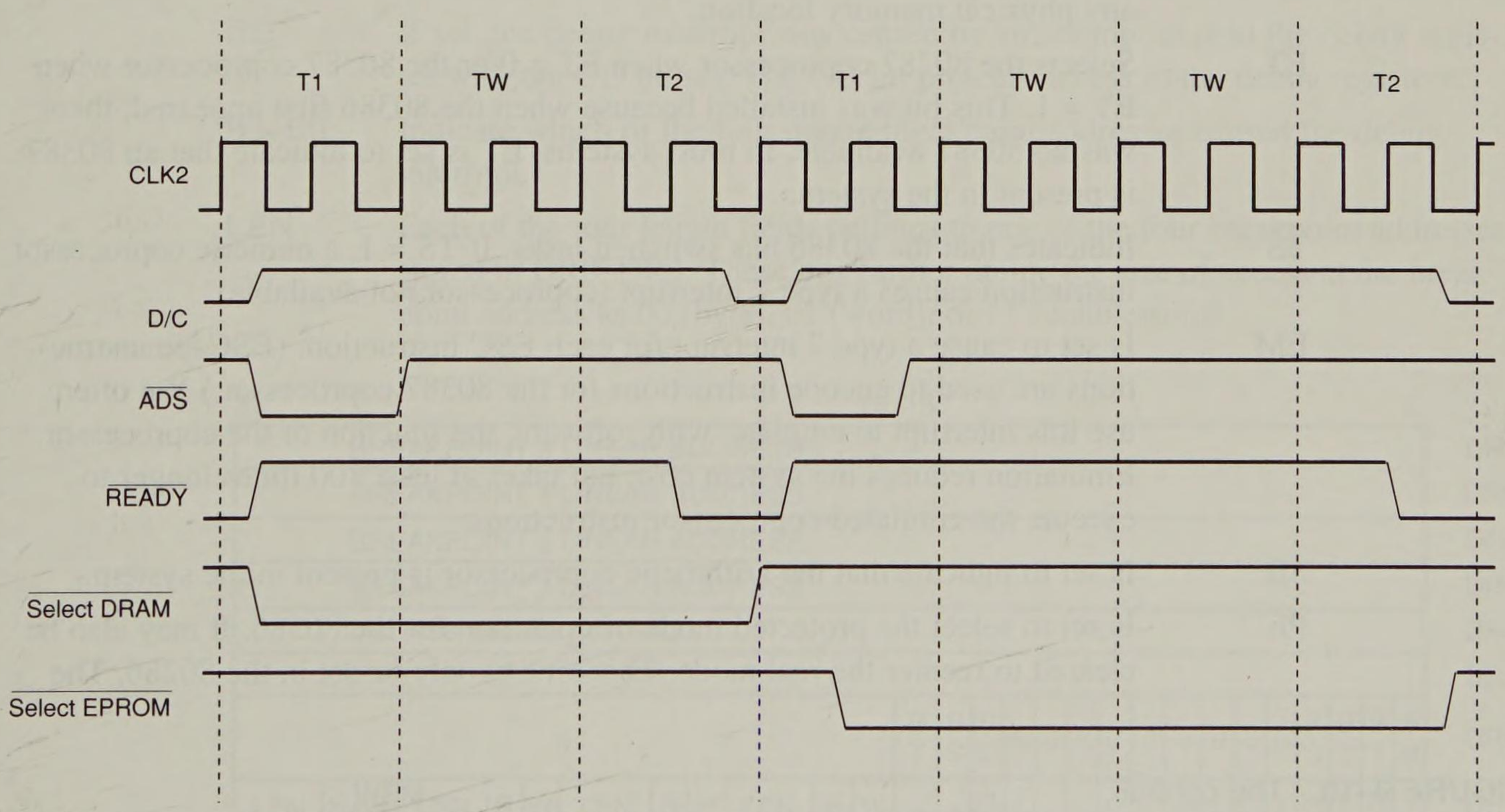
The $\overline{\text{READY}}$ signal is sampled at the end of a bus cycle to determine whether the clock cycle is T2 or TW. If $\overline{\text{READY}}$ = 0 at this time, it is the end of the bus cycle, or T2. If $\overline{\text{READY}}$ is 1 at the end of a clock cycle, it is a TW cycle, and the microprocessor continues to test $\overline{\text{READY}}$ searching for a logic 0 and the end of the bus cycle.

In the nonpipelined system, whenever $\overline{\text{ADS}}$ becomes a logic 0, a wait state is inserted if $\overline{\text{READY}}$ = 1. After $\overline{\text{ADS}}$ returns to a logic 1, the positive edges of the clock are counted to generate the $\overline{\text{READY}}$ signal. The $\overline{\text{READY}}$ signal becomes a logic 0 after the first clock to insert 0 waits states. If 1 wait state is inserted, the READY line must remain a logic 1 until at least two clocks have elapsed. If additional wait states are desired, then additional time must elapse before $\overline{\text{READY}}$ is cleared. This allows any number of wait states to be inserted into the timing.

Figure 9–9 shows a circuit that inserts 0 through 3 wait states for various memory addresses. In the example, 1 wait state is produced for a DRAM access and 2 waits for an EPROM access. The 74F164 clears whenever $\overline{\text{ADS}}$ is low and D/$\overline{\text{C}}$ is high. It begins to shift after $\overline{\text{ADS}}$ returns to a logic 1 level. As it shifts the 00000000, the shift register begins to fill with logic 1s from the QA connection towards the QH connection. The 4 different outputs are connected to an inverting multiplexer that generates the active low $\overline{\text{READY}}$ signal.




(a)




(b)

**FIGURE 9–9** (a) Circuit and (b) timing that selects 1 wait state for DRAM and 2 waits for EPROM.

## 9–2 SPECIAL 80386 REGISTERS

New registers, not found in earleir Intel microprocessors, appear in the 80386: control, debug, and test registers. Registers CR0–CR3 control various features, registers DR0–DR7 facilitate debugging, and registers TR6 and TR7 are used to test paging and caching.

## Control Registers

In addition to EFLAGS and EIP, described earlier, there are other control registers found in the 80386. Control register 0 (CR0) is identical to the MSW (machine status word) found in the 80286 microprocessor, except it is 32 bits, instead of 16 bits, wide. Additional control registers are CR1, CR2, and CR3.

Figure 9–10 illustrates the control register structure of the 80386. Control register CR1 is not used in the 80386, but is reserved for future products. Control register CR2 holds the linear page address of the last page accessed before a page fault interrupt. Control register CR3 holds the base address of the page directory. The right-most 12 bits of the 32-bit page table address contain zeros and combine with the remainder of the register to locate the start of the 4K-long page table.

Register CR0 contains a number of special control bits that are defined as follows in the 80386:

PG — Selects page table translation of linear addresses into physical addresses when PG = 1. Page table translation allows any linear address to be assigned any physical memory location.

ET — Selects the 80287 coprocessor when ET = 0 or the 80387 coprocessor when ET = 1. This bit was installed because when the 80386 first appeared, there was no 80387 available. In most systems, ET is set to indicate that an 80387 is present in the system.

TS — Indicates that the 80386 has switched tasks. If TS = 1, a numeric coprocessor instruction causes a type 7 interrupt (coprocessor not available).

EM — Is set to cause a type 7 interrupt for each ESC instruction. (ESCape instructions are used to encode instructions for the 80387 coprocessor.) We often use this interrupt to emulate, with software, the function of the coprocessor. Emulation reduces the system cost, but takes at least 100 times longer to execute the emulated coprocessor instructions.

MP — Is set to indicate that the arithmetic coprocessor is present in the system.

PE — Is set to select the protected mode of operation for the 80386. It may also be cleared to reenter the real mode. This bit can only be set in the 80286. The

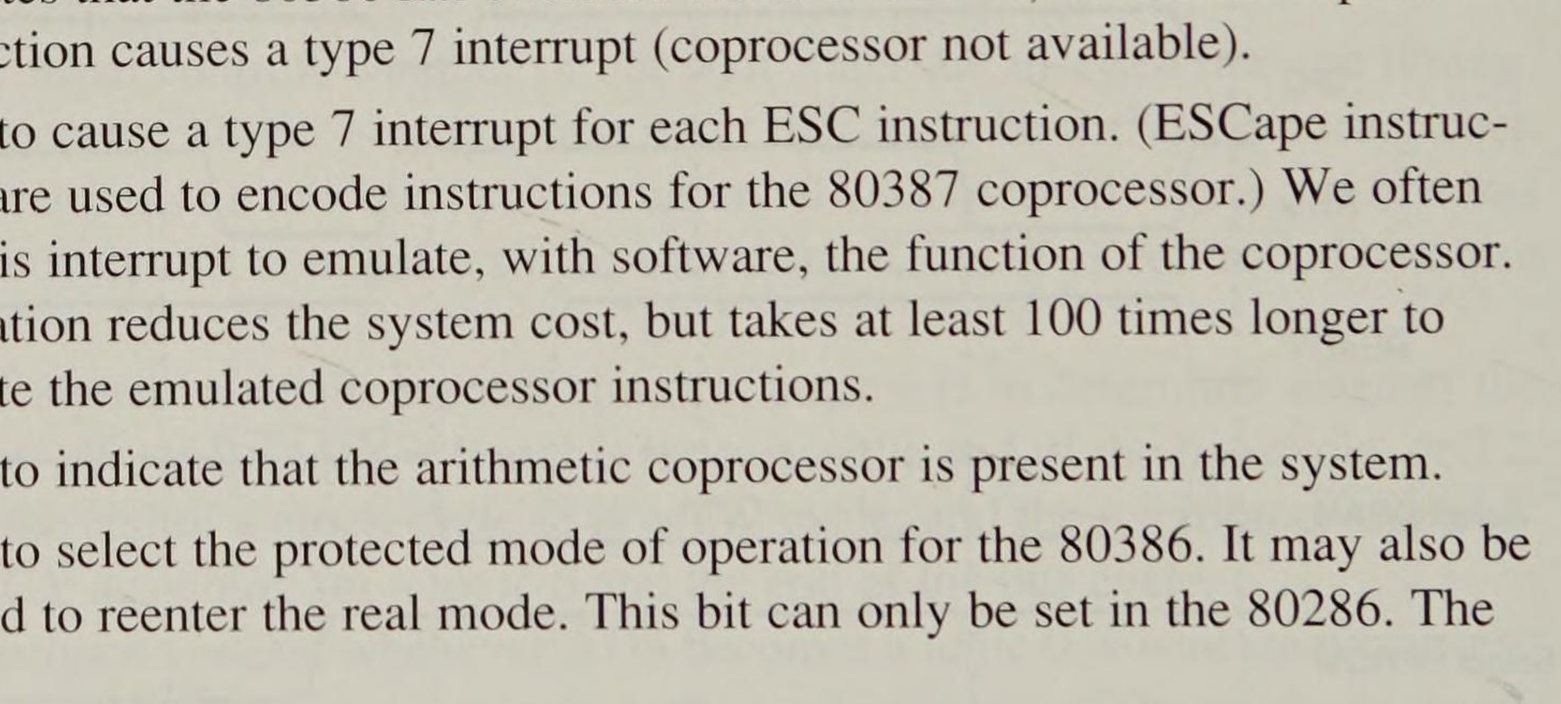



**FIGURE 9–10** The control-register structure of the 80386 microprocessor.

80286 cannot return to real mode without a hardware reset, which precludes its use in most systems that operate in the protected mode.

## Debug and Test Registers

Figure 9–11 shows the sets of debug and test registers. The first four debug registers contain 32-bit linear breakpoint addresses. (A *linear address* is a 32-bit address generated by a microprocessor instruction, that may or may not be the same as the physical address.) The breakpoint addresses, which may locate an instruction or datum, are constantly compared with the addresses generated by the program. If a match occurs, the 80386 will, if directed by debug registers DR6 and DR7, cause a type 1 interrupt (TRAP or debug interrupt) to occur. This feature is a much-expanded version of the basic trapping or tracing allowed with the earlier Intel microprocessors through the type 1 interrupt. The breakpoint addresses are very useful in debugging faulty software. The control bits in DR6 and DR7 are defined as follows:

BT If set, the debug interrupt was caused by a task switch.

BS If set, the debug interrupt was caused by the TF bit in the flag register.

BD If set, the debug interrupt was caused by an attempt to read the debug register with the GD bit set. The GD bit protects access to the debug registers.

B3–B0 Indicate which of the four debug breakpoint addresses caused the debug interrupt.

LEN Each of the four length fields pertains to one of the four breakpoint addresses stored in DR0–DR3. These bits further define the size of access at the breakpoint address as 00 (byte), 01 (word), or 11 (doubleword).

| 31 16 | 15 0 | |
|---|---|---|
| BREAKPOINT 0 LINEAR ADDRESS | | DR0 |
| BREAKPOINT 1 LINEAR ADDRESS | | DR1 |
| BREAKPOINT 2 LINEAR ADDRESS | | DR2 |
| BREAKPOINT 3 LINEAR ADDRESS | | DR3 |
| Intel reserved. Do not define. | | DR4 |
| Intel reserved. Do not define. | | DR5 |

| 31 16 | 15 14 13 12 | 11 10 9 8 | 7 6 5 4 | 3 2 1 0 | |
|---|---|---|---|---|---|
| 0 | BT BS BD 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 | B3 B2 B1 B0 | DR6 |
| LEN3 R3 W3 LEN2 R2 W2 LEN1 R1 W1 LEN0 R0 W0 | 0 0 | GD 0 0 0 | GE LE G3 L3 G2 L2 G1 L1 G0 L0 | | DR7 |

| 31 12 | 11 10 9 8 7 6 5 4 3 2 1 0 | |
|---|---|---|
| LINEAR ADDRESS | V D D# U U# W W# 0 0 0 0 C | TR6 |
| PHYSICAL ADDRESS | 0 0 0 0 0 0 0 PL REP 0 0 | TR7 |

**FIGURE 9–11** The debug and test registers of the 80386. (Courtesy of Intel Corporation)

RW — Each of the four read/write fields pertains to one of the four breakpoint addresses stored in DR0–DR3. The RW field selects the cause of action that enabled a breakpoint address as 00 (instruction access), 01 (data write), or 11 (data read and write).

GD — If set, GD prevents any read or write of a debug register by generating the debug interrupt. This bit is automatically cleared during the debug interrupt, so the debug registers can be read or changed if needed.

GE — If set, GE selects a global breakpoint address for each of the four breakpoint address registers.

LE — If set, LE selects a local breakpoint address for each of the four breakpoint address registers.

The test registers, TR6 and TR7, are used to test the *translation lookaside buffer* (TLB). The TLB is used with the paging unit within the 80386. The TLB holds the most commonly used page table address translations. The TLB reduces the number of memory reads required for looking up page translation addresses in the page translation tables. The TLB holds the most common 32 entries from the page table, and it is tested with the TR6 and TR7 test registers.

Test register TR6 holds the tag field (linear address) of the TLB, and TR7 holds the physical address of the TLB. To write a TLB entry, perform the following steps:

1. Write TR7 for the desired physical address, PL, and REP values.
2. Write TR6 with the linear address, making sure that C = 0.

To read a TLB entry:

1. Write TR6 with the linear address, making sure that C = 1.
2. Read both TR6 and TR7. If the PL bit indicates a hit, then the desired values of TR6 and TR7 indicate the contents of the TLB.

The bits found in TR6 and TR7 indicate the following conditions:

V — Shows the entry in the TLB is valid.

D — Indicates the entry in the TLB is invalid or dirty.

U — A user bit for the TLB.

W — Indicates that the area addressed by the TLB entry is writable.

C — Selects a write (0) or immediate lookup (1) for the TLB.

PL — Indicates a hit if a logic 1.

REP — Selects which block of the RLB is written.

Refer to the sections on the memory management and paging units for more detail on the function of the TLB.

## 9–3 80386 MEMORY MANAGEMENT

The memory management unit (MMU) in the 80386 is similar to the MMU in the 80286, except the 80386 has a paging unit not found in the 80286. The MMU performs the task

of converting linear addresses, as they appear as outputs from a program, into physical addresses that access a physical memory location located anywhere within the memory system. The 80386 uses the paging mechanism to allocate any physical address to any logical address. Therefore, even though the program is accessing memory location A0000H with an instruction, the actual physical address could be memory location 100000H or any other location, if paging is enabled. This feature allows virtually any software, written to operate at any memory location, to function in an 8038, because any linear location can become any physical location. Earlier Intel microprocessors did not have this flexibility. Paging is used with DOS 5.0/6.X to relocate 80386 and 80486 memory to addresses above FFFFFH (extended memory) into spaces between ROMs at locations D0000–DFFFFH and other areas as they are available (upper memory).

## Descriptors and Selectors

Before the memory management unit is discussed, the descriptor and selector for the 80386 microprocessor must be examined. The 80386 uses descriptors in much the same way as the 80286. In both microprocessors, a *descriptor* is a series of 8 bytes that describe and locate a memory segment. A *selector* (segment register) is used to index a descriptor from a table of descriptors. The main difference between the 80286 and 80386 is that the 80386 has two additional selectors (FS and GS), and the most significant two bytes of the descriptor are defined. Another difference is that 80386 descriptors use a 32-bit base address and a 20-bit limit instead of a 24-bit base address and a 16-bit limit as found on the 80286.

The 80286 addresses a 16M byte memory space with its 24-bit base address and has a segment length limit of 64K bytes due to the 16-bit limit. The 80386 addresses a 4G byte memory space with its 32-bit base address and has a segment length limit of 1M bytes, or 4G bytes, due to a 20-bit limit. The 20-bit limit is used in two different ways. If the granularity bit (G) = 0, the 20-bit limit can access a segment with a length of 1M byte. If G = 1, the 20-bit limit allows a segment length of 4G bytes.

The granularity bit is found in the 80386 descriptor. If G = 0, the number stored in the limit is interpreted, allowing it to contain any limit between 00000H and FFFFFH, for a segment size up to 1M bytes. If G = 1, the number stored in the limit is interpreted as 00000XXXH through FFFFFXXXH, where the XXX is 000H. This allows the limit of the segment to range from 0 bytes to 4G bytes, in steps of 4K bytes. A limit of 00001H indicates that the limit is 4K bytes when G = 1, and 1 byte when G = 0. An example is a segment that begins at physical address 10000000H. If the limit is 00001H and G = 0, this segment begins at 10000000H and ends at 10000001H. If G = 1 with the same limit (00001H), the segment begins at location 10000000H and ends at location 10001000H.

Figure 9–12 shows how the 80386 addresses a memory segment in the protected mode using a selector and a descriptor. Note that this is identical to the way that a segment is addressed by the 80286. The difference is the size of the segment addressed by the 80386. The selector uses its left-most 13 bits to select a descriptor from a descriptor table. The TI bit indicates either the local (TI = 1) or global (TI = 0) descriptor table. The right-most 2 bits of the selector define the requested privilege level of the access.

Because the selector uses a 13-bit code to access a descriptor, there are, at most, 8,192 descriptors in each table, local or global. Since each segment in an 80386 can be 4G bytes in length, we can access 16,384 segments at a time with the two descriptor

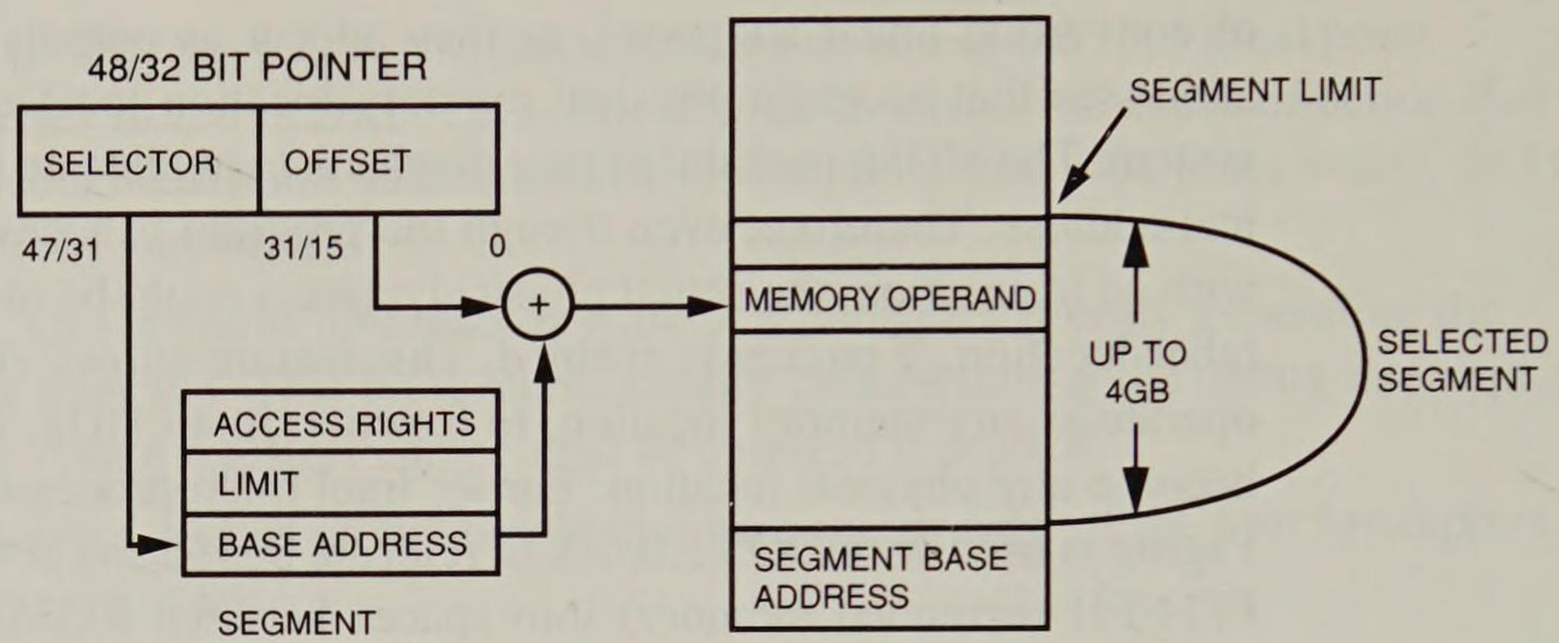



**FIGURE 9–12** Protected-mode addressing using a segment register as a selector. (Courtesy of Intel Corporation)

tables. This allows the 80386 to access a virtual memory size of 64T bytes. Of course, only 4G bytes of memory actually exist in the memory system. (1T byte = 1,024G bytes.) If a program requires more than 4G bytes of memory at a time, it can be swapped between the memory system and a disk drive or other form of large-volume storage.

The 80386 uses descriptor tables for both global (GDT) and local (LDT) descriptors. A third descriptor table appears for interrupt (IDT) descriptors or gates. The first six bytes of the descriptor are the same as in the 80286, which allows 80286 software to be upward compatible with the 80386. (An 80286 descriptor used 00H for its most significant two bytes.) See Figure 9–13 for the 80286 and 80386 descriptors. The base address is 32 bits in the 80386, the limit is 20 bits, and a G bit selects the limit multiplier (1 or 4K). The fields for the 80386 descriptor are defined as follows:

| | |
|---|---|
| Base (B0–B31) | Defines the starting 32-bit address of the segment within the 4G byte physical address space of the 80386 microprocessor. |
| Limit (L0–L19) | Defines the limit of the segment in units of bytes if the G bit = 0, or in units of 4K bytes if G = 1. This allows a segment to be of any length from 1 byte to 1M bytes if G = 0, and from 4K bytes to 4G bytes if G = 1. Recall that the limit indicates the last byte in a segment. |
| Access Rights | Determines privilege level and other information about the segment. This byte varies with different types of descriptors and is elaborated with each descriptor type. |

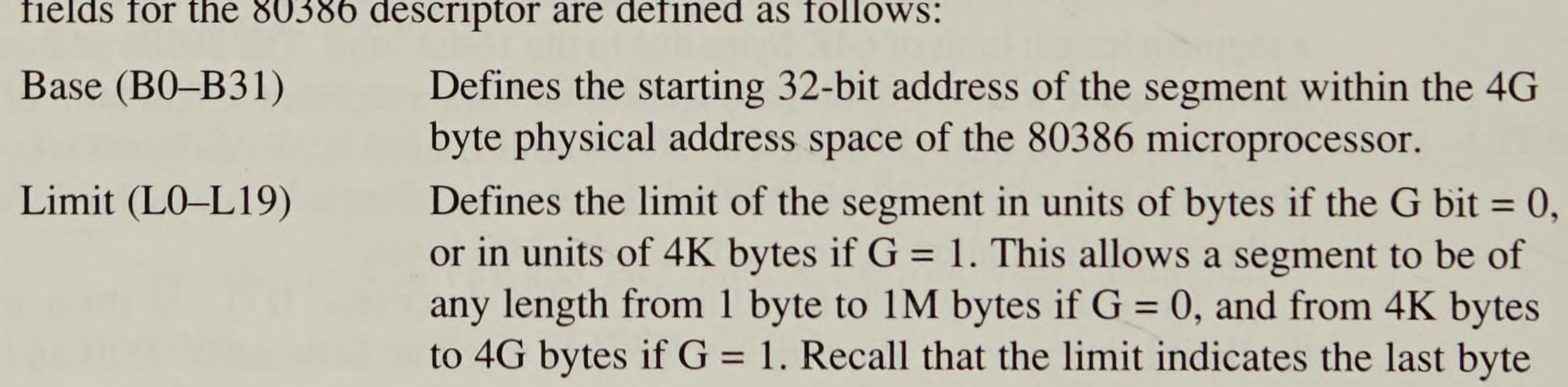



**FIGURE 9–13** The descriptors for the 80286 and 80386 microprocessors.

G — The granularity bit selects a multiplier of 1 or 4K for the limit field. If G = 0, the multiplier is 1 and if G = 1, the multiplier is 4K.

D — Selects the default register size. If D = 0, the registers are 16 bits wide, as in the 80286. If D = 1, the registers are 32 bits wide, as in the 80386. This bit determines whether prefixes are required for 32-bit data and index registers. If D = 0, then a prefix is required to access 32-bit registers and to use 32-bit pointers. If D = 1, then a prefix is required to access 16-bit registers and 16-bit pointers. The USE16 and USE32 directives appended to the SEGMENT statement in assembly language control the setting of the D bit. In the real mode, it is always assumed that the registers are 16 bits wide so any instruction that references a 32-bit register or pointer must be prefixed. Note that current versions of DOS and OS2* assume that D = 0.

AVL — This bit is available to the operating system to use in any way it sees fit. It often indicates that the segment described by the descriptor is available.

Descriptors appear in two forms in the 80386 microprocessor: the segment descriptor and the system descriptor. The segment descriptor defines data, stack, and code segments and the system descriptor defines information about the system's tables, tasks, and gates.

*Segment Descriptors.* Figure 9–14 shows the segment descriptor. This descriptor fits the general form dictated in Figure 9–13, but the access rights bits are defined to indicate how the data, stack, or code segment described by the descriptor functions. Bit position 4 of the access rights byte determines whether the descriptor is a data or code segment descriptor (S = 1), or a system segment descriptor (S = 0). Note that the labels used for these bits may vary in different versions of Intel literature, but perform the same tasks.

Following is a description of the access rights bits and their function in the segment descriptor:

P — (Present)—A logic 1 indicates that the segment is present. If P = 0 and the segment is accessed through the descriptor, a type 11 interrupt occurs. This

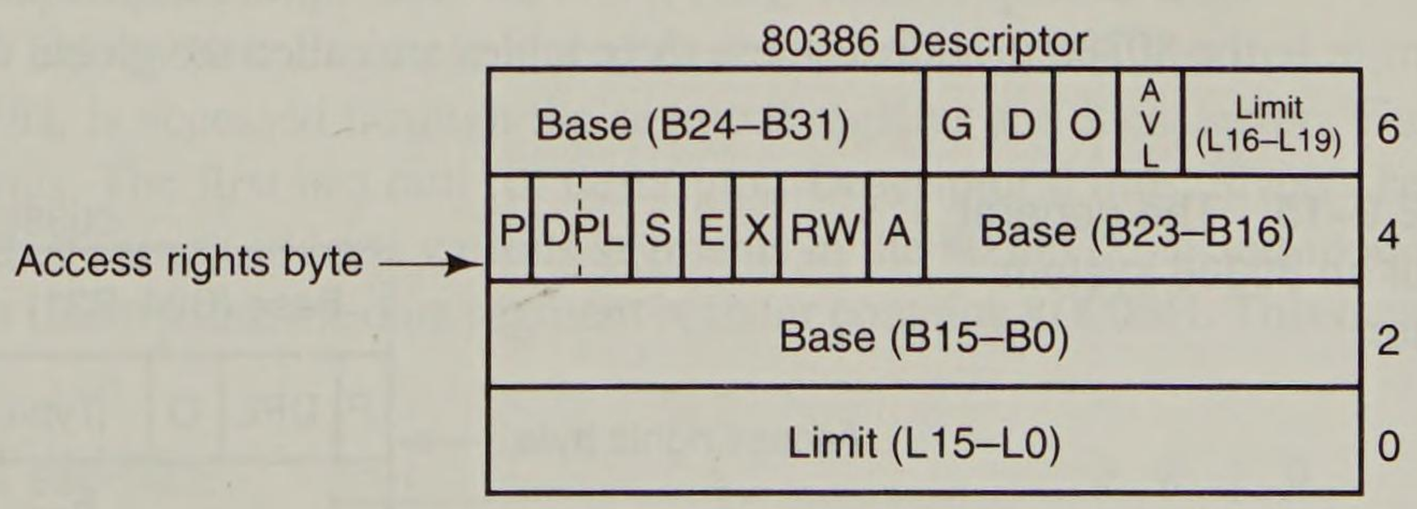



**FIGURE 9–14** The format of the 80386 segment descriptor.

*OS2 (Operating System 2) is a registered trademark of IBM Inc.

interrupt indicates that a segment that is not present in the system was accessed.

DPL (Descriptor Privilege Level)—Sets the privilege level of the descriptor, where 00 has the highest privilege and 11 has the lowest. This is used to protect access to segments. If a segment is accessed with a privilege level that is lower (higher in number) than the DPL, a privilege violation interrupt occurs. Privilege levels are used in multi-user systems to prevent access to an area of the system memory.

S (Segment)—Indicates a data or code segment descriptor (S = 1) or a system segment descriptor (S = 0).

E (Executable)—Selects a data (stack) segment (E = 0) or a code segment (E = 1). E also defines the function of the next two bits (X and RW).

X If E = 0, then X indicates the direction of expansion for the data segment. If X = 0, the segment expands upward as in a data segment. If X = 1, the segment expands downward as in a stack segment. If E = 1, then X indicates whether the privilege level of the code segment is ignored (X = 0) or observed (X = 1).

RW (Read/Write)—If E = 0, then RW indicates that the data segment may be written (RW = 1) or not written (RW = 0). If E = 1, then RW indicates that the code segment may be read (RW = 1) or not read (RW = 0).

A (Accessed)—This bit is set each time the microprocessor accesses the segment. It is sometimes used by the operating system to keep track of which segments have been accessed.

*System Descriptor.* The system descriptor is illustrated in Figure 9–15. There are 16 possible system descriptor types (see Table 9–2 for the different descriptor types), but not all are used in the 80386 microprocessor. Some of these types are defined for the 80286, so the 80286 software is compatible with the 80386. Some of the types are new and unique to the 80386. Some have yet to be defined and are reserved for future Intel products.

## Descriptor Tables

The descriptor tables define all the segments used in the 80386 when operated in the protected mode. There are three types of descriptor tables: the global descriptor table (GDT), the local descriptor table (LDT), and the interrupt descriptor table (IDT). The registers used by the 80386 to address these three tables are called the global descriptor table register (GDTR),

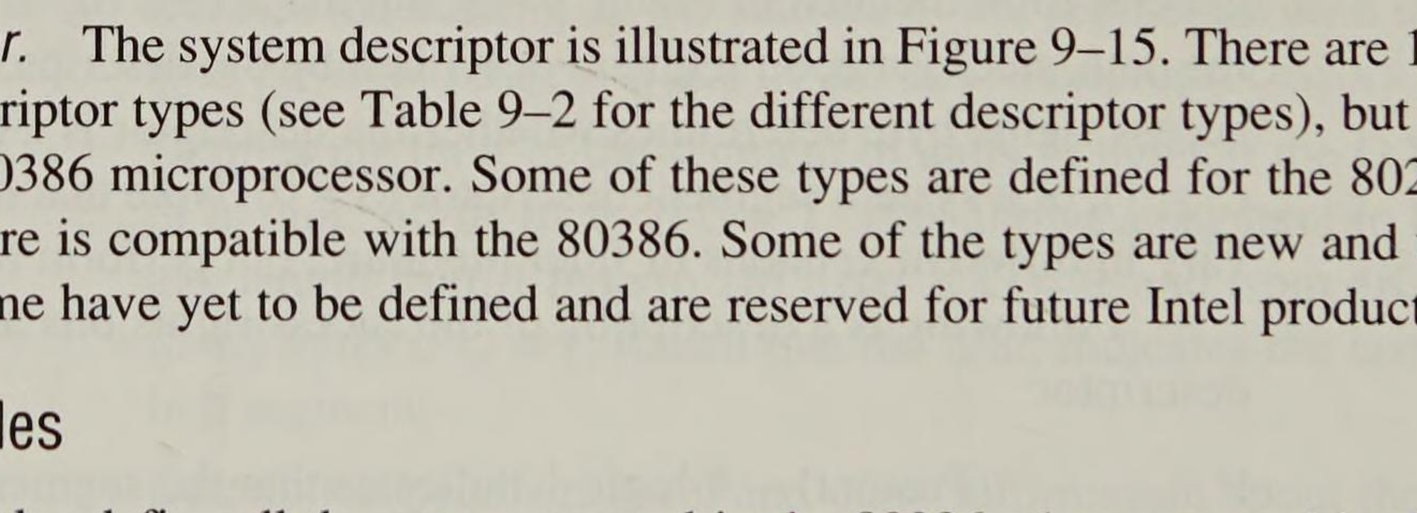



**FIGURE 9–15** The general format of an 80386 system descriptor.

**TABLE 9–2** 80386 system descriptor types

| *Type* | *Description* |
|---|---|
| 0000 | Invalid |
| 0001 | Available 80286 TSS |
| 0010 | LDT |
| 0011 | Busy 80286 TSS |
| 0100 | 80286 call gate |
| 0101 | Task gate (80286 or 80386) |
| 0110 | 80286 interrupt gate |
| 0111 | 80286 trap gate |
| 1000 | Invalid |
| 1001 | Available 80386 TSS |
| 1010 | Reserved for future Intel products |
| 1011 | Busy 80386 TSS |
| 1100 | 80386 call gate |
| 1101 | Reserved for future Intel products |
| 1110 | 80386 interrupt gate |
| 1111 | 80386 trap gate |

the local descriptor table register (LDTR), and the interrupt descriptor table register (IDTR). These registers are loaded respectively with the LGDT, LLDT, and LIDT instructions.

The *descriptor table* is a variable-length array of data, with each entry holding an 8-byte descriptor. The local and global descriptor tables hold up to 8,192 entries each, and the interrupt descriptor table holds up to 256 entries. A descriptor is indexed from either the local or global descriptor table by the selector that appears in a segment register. Figure 9–16 shows a segment register, and the selector that it holds, in the protected mode. The left-most 13 bits index a descriptor, the TI bit selects either the local (TI = 1) or global (TI = 0) descriptor table, and the RPL bits indicate the requested privilege level.

Whenever a new selector is placed into one of the segment registers, the 80386 accesses one of the descriptor tables and automatically loads it into a program-invisible cache portion of the segment register. As long as the selector remains the same in the segment register, no additional accesses are required to the descriptor table. The operation of fetching a new descriptor from the descriptor table is program-invisible, because the microprocessor automatically accomplishes this each time the segment register contents are changed in the protected mode.

Figure 9–17 shows how a sample global descriptor table (GDT), stored at memory address 00010000H, is accessed through the segment register and its selector. This table contains four entries. The first is a null (0) descriptor. Descriptor 0 must always be a null descriptor. The other entries address various segments in the 80386 protected-mode memory system. In this illustration, the data segment register contains a 0008H. This means that

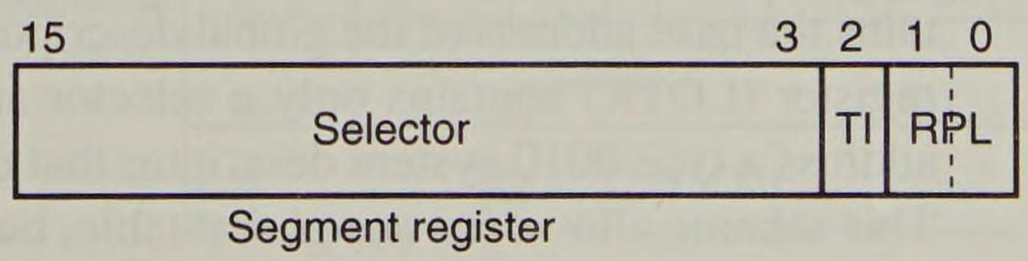



**FIGURE 9–16** A segment register showing the selector, TI bit, and requested privilege level (RPL) bits.

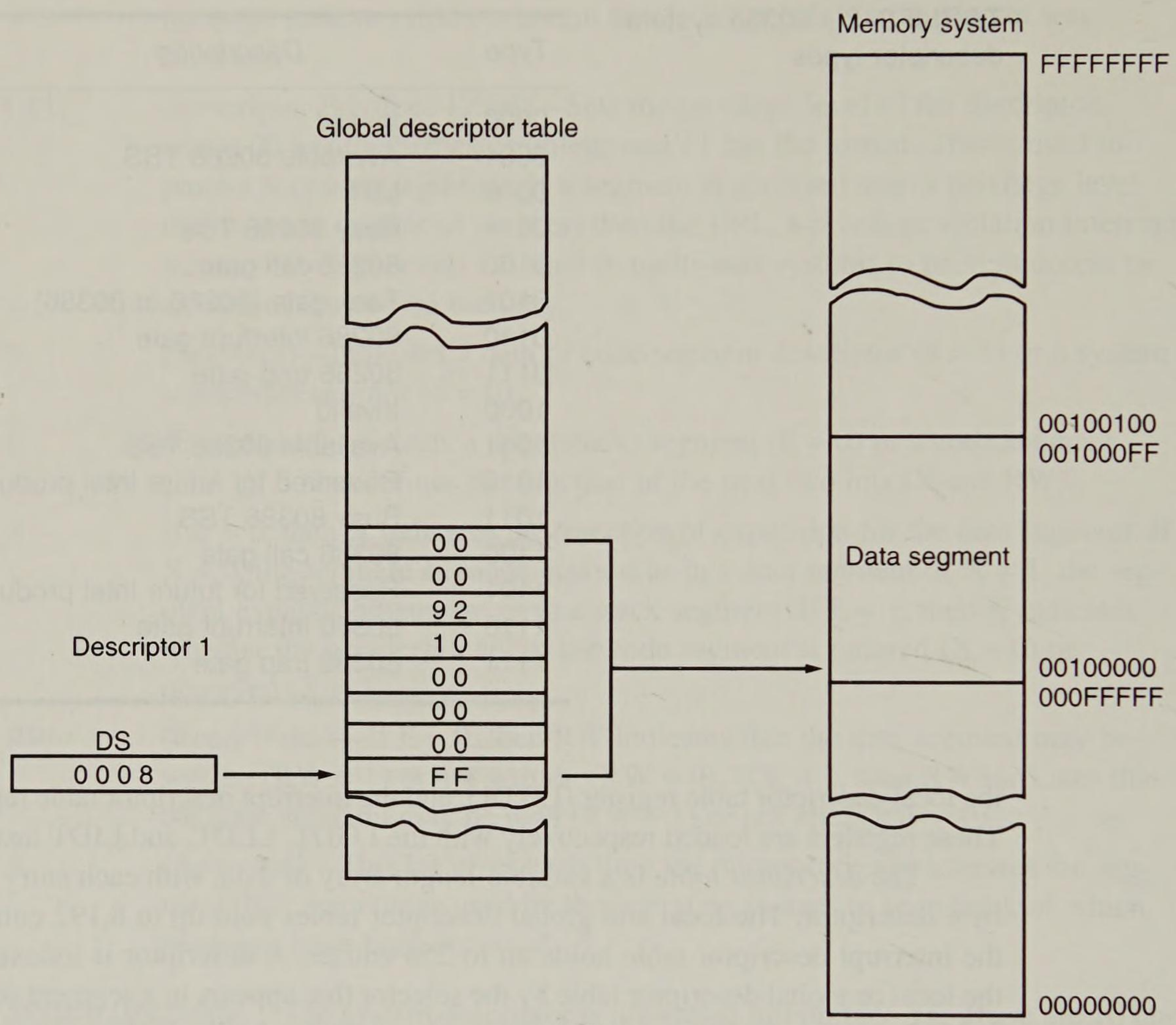



**FIGURE 9–17** Using the DS register to select a descriptor from the Global Descriptor Table. In this example, the DS register accesses memory locations 00100000H–001000FFH as a data segment.

the selector is indexing descriptor location 1 in the global descriptor table (TI = 0), with a requested privilege level of 00. Descriptor 1 is located 8 bytes above the base descriptor table address, beginning at location 00010008H. The descriptor located in this memory location accesses a base address of 00200000H and a limit of 100H. This means that this descriptor addresses memory locations 00200000H–00200100H. Because this is the DS (data segment) register, this means that the data segment is located at these locations in the memory system. If data are accessed outside of these boundaries, an interrupt occurs.

The local descriptor table (LDT) is accessed in the same manner as the global descriptor table (GDT). The main difference in access is that the TI bit is cleared for global access and set for local access. Another difference is apparent if the local and global descriptor table registers are examined. The global descriptor table register (GDTR) contains the base address of the global descriptor table and the limit. The local descriptor table register (LDTR) contains only a selector and is 16 bits wide. The contents of the LDTR address a type 0010 system descriptor that contains the base address and limit of the LDT. This scheme allows for one global table, but many local tables, one or more for each task,

if necessary. Global descriptors describe memory for the system, while local descriptors describe memory for applications or tasks.

The interrupt descriptor table (IDT), like the GDT, is addressed by storing the base address and limit in the interrupt descriptor table register (IDTR). The main difference between the GDT and IDT is that the IDT contains only interrupt gates, while the GDT and LDT also contain segment and system descriptors.

Figure 9–18 shows the gate descriptor, a special form of the system descriptor described earlier. (Refer to Table 9–2 for the different gate descriptor types.) Notice that the gate descriptor contains a 32-bit offset address, a word count, and a selector. The 32-bit offset address points to the location of the interrupt service procedure or other procedure. The word count indicates how many words are transferred from the caller's stack to the stack of the procedure accessed by a call gate. Note that the word count field is not used with an interrupt gate. The selector is used to indicate the location of the task state segment (TSS) in the GDT or LDT if it is a local procedure.

When a gate is accessed, the contents of the selector are loaded into the task register (TR). The acceptance of the gate depends on the privilege and priority levels. A return instruction (RET) ends a call gate procedure, and a return from interrupt instruction (IRET) ends an interrupt gate procedure. Tasks are usually accessed with a CALL or an INT instruction, where the call instruction addresses a call gate in the descriptor table and the interrupt instruction addresses an interrupt descriptor.

The difference between real-mode interrupts and protected-mode interrupts is that the interrupt vector table is an IDT in the protected mode. The IDT still contains up to 256 interrupt levels, but each level is accessed through an interrupt gate instead of an interrupt vector. Thus, interrupt type number 2 is located at IDT descriptor number 2, 16 locations above the base address of the IDT. This means that the first 1K byte of memory no longer contains interrupt vectors as it did in the real mode. The IDT can be located at any address in the memory system.

## The Task State Segment (TSS)

The task state segment (TSS) descriptor contains information about the location, size, and privilege level of the task state segment, just like any other descriptor. The difference is that the task state segment described by the TSS descriptor does not contain data or code. It contains the state of the task and linkage so tasks can be nested; i.e., one task can call a second, which can call a third, and so forth. The TSS descriptor is addressed by the task register (TR). The contents of the TR are changed by the LTR instruction or whenever the protected-mode program executes a far JMP or CALL instruction. The LTR instruction is used to access a

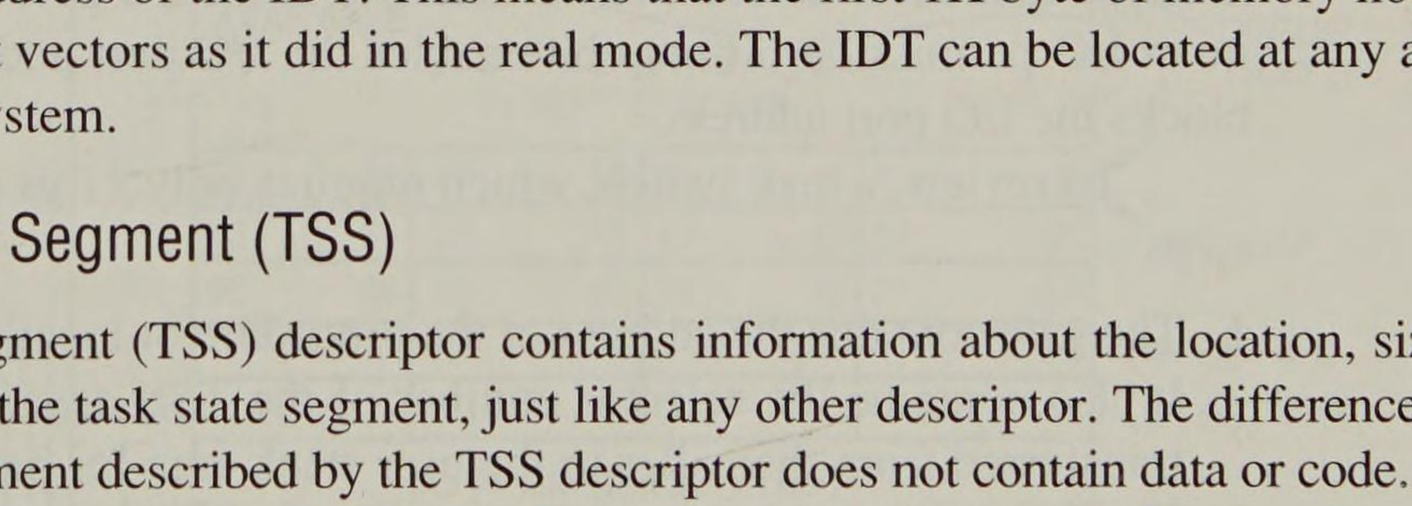



**FIGURE 9–18** The gate descriptor for the 80386 microprocessor.

task during system initialization. After initialization, the CALL or JMP instructions normally switch tasks. In most cases we use the CALL instruction to initiate a new task.

The TSS is illustrated in Figure 9–19. As can be seen, the TSS is a formidable section of memory, containing many different types of information. The first word of the TSS is labeled back-link. This is the selector that is loaded into the TR on a return (RET or IRET) to link back to the prior TSS. The following word must contain a 0. The second through the seventh doublewords contain the ESP and ESS values for privilege levels 0–2. These are required in case the current task is interrupted so these privilege level (PL) stacks can be addressed. The eighth word (offset 1CH) contains the contents of CR3, which stores the base address of the prior state's page directory register. This must be restored if paging is in effect. The contents of the next 17 doublewords are loaded into the registers indicated. Whenever a task is accessed, the state of the entire machine (all of the registers) is stored in these memory locations and then reloaded from the same locations in the new TSS. The last word (offset 66H) contains the I/O permission bit map base address.

The I/O permission bit map allows the TSS to block I/O operations to inhibited I/O port addresses via an I/O permission denial interrupt. The permission denial interrupt is type number 13, the general protection fault interrupt. The I/O permission bit map base address is the offset address from the start of the TSS. This allows the same permission map to be used by many TSSs.

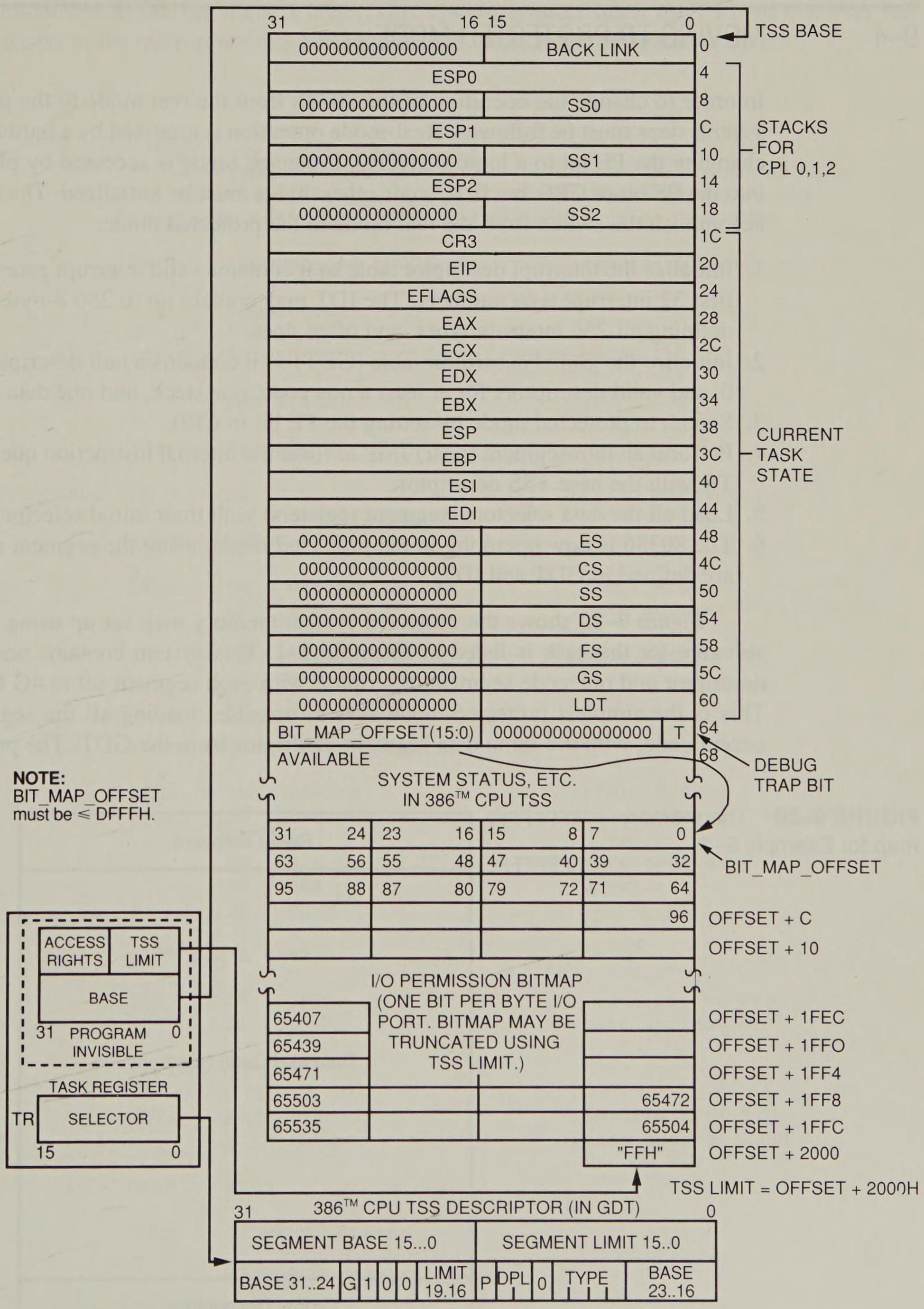
Each I/O permission bit map is 64K bits (8K bytes) in length, beginning at the offset address indicated by the I/O permission bit map base address. The first byte of the I/O permission bit map contains I/O permission for I/O ports 0000H–0007H. The right-most bit contains permission for port number 0000H, and the left-most bit contains permission for port number 0007H. This sequence continues until the very last port address (FFFFH) is stored in the left-most bit of the last byte of the I/O permission bit map. A logic 0 placed in an I/O permission bit map bit enables the I/O port address, while a logic 1 inhibits or blocks the I/O port address.

To review, a task switch, which requires only 17 μs to execute, entails the following steps:

1. The gate contains the address of the procedure or location jumped to by the task switch. It also contains the selector number of the TSS descriptor, and the number of words transferred from the caller to the user stack area for parameter passing.
2. The selector is loaded into TR from the gate. (This step is accomplished by a CALL or JMP that refers to a valid TSS descriptor.)
3. The TR selects the TSS.
4. The current state is saved in the current TSS and the new TSS is accessed with the state of the new task (all the registers) loaded into the microprocessor. The current state is saved at the TSS selector currently found in the TR. Once the current state is saved, a new value for the TSS selector is loaded into TR (by the JMP or CALL) and the new state is loaded from the new TSS.

The return from a task is accomplished by the following steps:

1. The current state of the microprocessor is saved in the current TSS.
2. The back-link selector is loaded to the TR to access the prior TSS, so the prior state of the machine can be restored to the microprocessor. The return for a called TSS is accomplished by the IRET instruction.




Type = 9: Available 386™ CPU TSS.
Type = B: Busy 386™ CPU TSS.

**FIGURE 9–19** The task state segment (TSS) descriptor. (Courtesy of Intel Corporation)

## 9–4 MOVING TO PROTECTED MODE

In order to change the operation of the 80386 from the real mode to the protected mode, several steps must be followed. Real-mode operation is accessed by a hardware reset or by changing the PE bit to a logic 0 in CR0. Protected mode is accessed by placing a logic 1 into the PE bit of CR0, but first, some other things must be initialized. The following steps accomplish the switch from the real mode to the protected mode:

1. Initialize the interrupt descriptor table so it contains valid interrupt gates for at least the first 32 interrupt type numbers. The IDT may contain up to 256 8-byte interrupt gates defining all 256 interrupt types, and often does.
2. Initialize the global descriptor table (GDT) so it contains a null descriptor at descriptor 0, and valid descriptors for at least a one code, one stack, and one data segment.
3. Switch to protected mode by setting the PE bit in CR0.
4. Perform an intrasegment (near) JMP to flush the internal instruction queue and load the TR with the base TSS descriptor.
5. Load all the data selectors (segment registers) with their initial selector values.
6. The 80386 is now operating in the protected mode, using the segment descriptors that are defined in GDT and IDT.

Figure 9–20 shows the protected system memory map set up using steps 1–5. The software for this task is listed in Example 9–1. This system contains one data segment descriptor and one code segment descriptor, with each segment set to 4G bytes in length. This is the simplest protected-mode system possible, loading all the segment registers, except code, with the same data segment descriptor from the GDT. The privilege level is

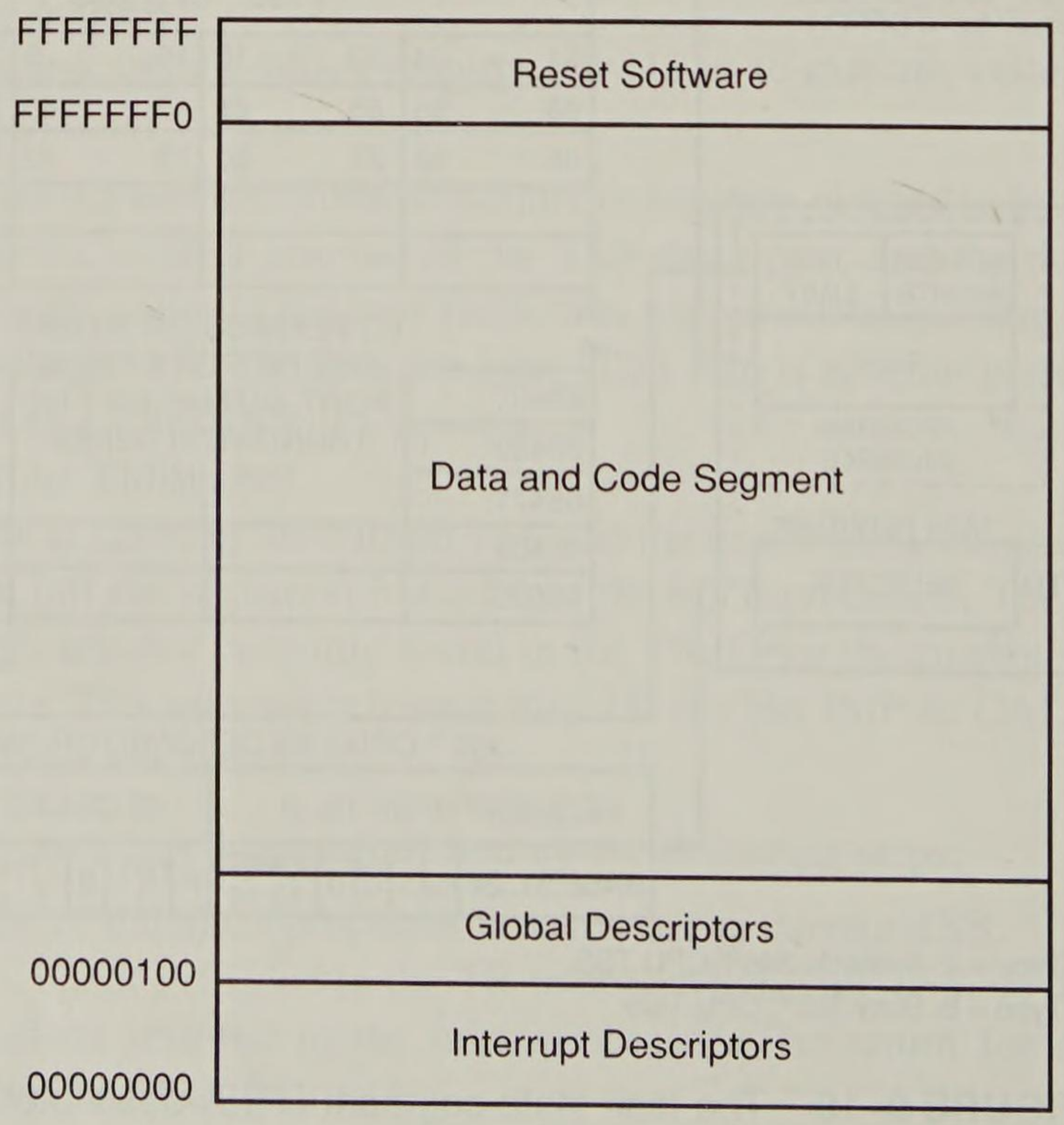



**FIGURE 9–20** The memory map for Example 9–3.

initialized to 00, the highest level. This system is most often used where one user has access to the microprocessor and requires the entire memory space.

**EXAMPLE 9–1**

```
                        .386p
                        ;This software causes the 80386 to enter the protected mode.
                        ;It does not illustrate the interrupt descriptors or any
                        ;software that will be executed in the protected mode.

0000                    DATA    SEGMENT  AT 0000H              ;segment address = 0000H

0000                            ORG 0000H

                        ;first 32 interrupt vectors placed here

0100                            ORG 0100H

                        ;Global Descriptor Table

0100 0000000000000000   DES0    DQ  0                         ;null descriptor

                        ;Code Segment Descriptor

0108 FFFF               DES1    DW   0FFFFH                   ;limit 4 G
010A 0000                       DW   0                        ;base address = 00000000H
010C 00                         DB   0
010D 9E                         DB   9EH                      ;code segment
010E 8F                         DB   8FH                      ;G = 1
010F 00                         DB   0

                        ;Data Segment Descriptor

0110 FFFF               DES2    DW   0FFFFH                   ;limit 4 G
0112 0000                       DW   0                        ;base address = 00000000H
0114 00                         DB   0
0115 92                         DB   92H                      ;data segment
0116 8F                         DB   8FH                      ;G = 1
0117 00                         DB   0

                        ;IDT table data

0118 00FF               IDT     DW   0FFH                     ;set limit to FFH
011A 00000000                   DD   0                        ;base address 0H

                        ;GDT table data

011E 0017               GDT     DW   17H                      ;set limit to 17H
0120 00000100                   DD   100H                     ;base address 100H

0124                    DATA    ENDS

0000                    CODE    SEGMENT  USE16

                                ASSUME   CS:CODE,DS:DATA

0000 B8 ---- R          START:  MOV  AX,DATA                  ;load DS
0003 8E D8                      MOV  DS,AX

0005 67| 0F 01 1D 00000118 R   LIDT  FWORD PTR IDT           ;load IDTR
```

```
000D 67| 0F 01 15 0000011E R   LGDT FWORD PTR GDT          ;load GDTR

0015 0F 20 C0                  MOV  EAX,CR0               ;set PE
0018 0C 01                     OR   AL,1
001A 0F 22 C0                  MOV  CR0,EAX

001D EB 01 90                  JMP  START1                ;near jump

0020                   START1:

0020 B8 0010                   MOV  AX,10H                ;selector 2
0023 8E D8                     MOV  DS,AX
0025 8E C0                     MOV  ES,AX
0027 8E D0                     MOV  SS,AX
0029 8E E8                     MOV  GS,AX
002B 8E E0                     MOV  FS,AX
002D 66| BC FFFFFFFF           MOV  ESP,0FFFFFFFFH

                       ;at this point we are in the protected mode

0033                   CODE    ENDS

                               END  START
```

In more complex systems, the steps required to initialize the system in the protected mode are more involved. For complex, multi-user systems, the registers are loaded using the task state segment (TSS). The steps required to place the 80386 into protected-mode operation for a more complex system using a task switch are:

1. Initialize the interrupt descriptor table so it refers to valid interrupt descriptors with at least 32 descriptors in the IDT.
2. Initialize the global descriptor table so it contains at least two task state segment (TSS) descriptors, and the initial code and data segments required for the initial task.
3. Initialize the task register (TR) so it points to a valid TSS, because when the initial task switch occurs and accesses the new TSS, the current registers are stored in the initial TSS.
4. Switch to protected mode using an intrasegment (near) jump to flush the internal instruction queue. Load TR with the current TSS selector.
5. Load the TR with a far jump instruction to access the new TSS and save the current state.
6. The 80386 is now operating in the protected mode under control of the first task.

Example 9–2 illustrates the software required to initialize the system and switch to protected mode using a task switch. The initial system task operates at the highest level of protection (00) and controls the entire operating environment for the 80386. In many cases, it is used to boot (load) software that allows many users to access the system in a multi-user environment.

**EXAMPLE 9–2** (pp. 352–356)

```
0008                   DESC    STRUC                      ;define descriptor structure

0000 0000              LIMIT_L DW    0
0002 0000              BASE_L  DW    0
```

```
0004 00                 BASE_M  DB      0
0005 00                 ACCESS  DB      0
0006 00                 LIMIT_H DB      0
0007 00                 BASE_H  DB      0

                        DESC    ENDS

0068                    TSS     STRUC                   ;define TSS structure

0000 0000               BACK_L  DW      0
0002 0000                       DW      0
0004 00000000           ESP0    DD      0
0008 0000               SS0     DW      0
000A 0000                       DW      0
000C 00000000           ESP1    DD      0
0010 0000               SS1     DW      0
0012 0000                       DW      0
0014 00000000           ESP2    DD      0
0018 0000               SS2     DW      0
001A 0000                       DW      0
001C 00000000           CCR3    DD      0
0020 00000000           EIP     DD      0
0024 00000000           TFLAGS  DD      0
0028 00000000           EEAX    DD      0
002C 00000000           ECX     DD      0
0030 00000000           EEDX    DD      0
0034 00000000           EEBX    DD      0
0038 00000000           EESP    DD      0
003C 00000000           EEBP    DD      0
0040 00000000           EESI    DD      0
0044 00000000           EEDI    DD      0
0048 0020               EES     DW      20H
004A 0000                       DW      0
004C 0018               ECS     DW      18H
004E 0000                       DW      0
0050 0020               ESS     DW      20H
0052 0000                       DW      0
0054 0020               EDS     DW      20H
0056 0000                       DW      0
0058 0020               EFS     DW      20H
005A 0000                       DW      0
005C 0020               EGS     DW      20H
005E 0000                       DW      0
0060 0000               ELDT    DW      0
0062 0000                       DW      0
0064 0000                       DW      0
0066 0000               BIT_MAP DW      0

                        TSS     ENDS

0000                    STACK   SEGMENT STACK

0000 0400 [                     DW      400H DUP (?)
          0000
               ]

0800                    STACK   ENDS

                        .386P
0000                    CODE    SEGMENT USE16
```

```
                        ASSUME CS:CODE,SS:STACK


0000 0000 0000 00000000 TSS1    TSS  <>                      ;task state segment 1
     0000 0000 00000000
     0000 0000 00000000
     0000 0000 00000000
     00000000 00000000
     00000000 00000000
     00000000 00000000
     00000000 00000000
     0020 0000 0018
     0000 0020 0000
     0020 0000 0020
     0000 0020 0000
     0000 0000 0000
     0000

0068 0000 0000 00000000 TSS2    TSS  <>                      ;task state segment 2
     0000 0000 00000000
     0000 0000 00000000
     0000 0000 00000000
     00000000 00000000
     00000000 00000000
     00000000 00000000
     00000000 00000000
     0020 0000 0018
     0000 0020 0000
     0020 0000 0020
     0000 0020 0000
     0000 0000 0000
     0000
00D0 0800 [             IDT     DB      8*256 DUP (?)        ;interrupt descriptor table
          00
            ]
08D0 0000 0000 00 00    GDT     DESC <>                      ;null descriptor
     00 00
08D8 0000 0028 00 85    TG1     DESC    <0,28H,0,85H,0,0>    ;task gate 1
     00 00
08E0 0000 0030 00 85    TG2     DESC    <0,30H,0,85H,0,0>    ;task gate 2
     00 00
08E8 FFFF 0000 00 9     ACS     DESC    <-1,0,0,9AH,0CFH,0>  ;code segment (4G)
     CF 00
08F0 FFFF 0000 00 92    DS1     DESC    <-1,0,0,92H,0CFH,0>  ;data segment (4G)
     CF 00
08F8 FFFF 0000 00 89    TSS1    DESC    <-1,0,0,89H,0CFH,0>  ;TSS1 available
     CF 00
0900 FFFF 0000 00 89    TSS2    DESC    <-1,0,0,89H,0CFH,0>  ;TSS2 available
     CF 00
0908 2000 [             IOBP    DB      2000H DUP (0)        ;enable all I/O
          00
            ]
2908 FF                         DB      0FFH                 ;end of I/O bit map

0006                    GDT_A   STRUC

0000 0000               A       DW      0
0002 0000               B       DW      0
0004 0000               CC      DW      0
```

```
                        GDT_A   ENDS

2909 0000 0000 0000     GDT_AD  GDT_A   <>                      ;address of GDT

290F                    MAIN    PROC    FAR

290F 8C C8                      MOV     AX,CS
2911 8E D8                      MOV     DS,AX
2913 E8 0098                    CALL    DO_IDT                  ;set up IDT
2916 B8 0908 R                  MOV     AX,OFFSET IOBP          ;set up I/O bit maps
2919 2E: A3 0066 R              MOV     TSS1.BIT_MAP,AX
291D 2E: A3 00CE R              MOV     TSS2.BIT_MAP,AX

2921 66| 33 C0                 XOR     EAX,EAX                 ;get linear start address
2924 8C C8                      MOV     AX,CS
2926 66| C1 E0 04              SHL     EAX,4
292A 66| 33 DB                 XOR     EBX,EBX
292D BB 29AE R                  MOV     BX,OFFSET TASK1
2930 66| 50                    PUSH    EAX
2932 66| 03 C3                 ADD     EAX,EBX
2935 66| 2E: A3 0088 R         MOV     TSS2.EIP,EAX            ;setup TASK1 as start address
293A 66| 58                    POP     EAX
293C 66| 50                    PUSH    EAX
293E BB 0000 R                  MOV     BX,OFFSET TSS1          ;load TSS1 address
2941 66| 03 C3                 ADD     EAX,EBX
2944 2E: A3 0002 R              MOV     TSS1.BASE_L,AX
2948 66| C1 E8 10              SHR     EAX,16
294C 2E: A2 0004 R              MOV     TSS1.BASE_M,AL
2950 2E: 88 26 0007 R           MOV     TSS1.BASE_H,AH
2955 66| 58                    POP     EAX
2957 66| 50                    PUSH    EAX
2959 BB 0068 R                  MOV     BX,OFFSET TSS2          ;load TSS2 address
295C 66| 03 C3                 ADD     EAX,EBX
295F 2E: A3 006A R              MOV     TSS2.BASE_L,AX
2963 66| C1 E8 10              SHR     EAX,16
2967 2E: A2 006C R              MOV     TSS2.BASE_M,AL
296B 2E: 88 26 006F R           MOV     TSS2.BASE_H,AH

2970 66| 58                    POP     EAX
2972 B8 FFFF                    MOV     AX,-1
2975 2E: A3 290B R              MOV     GDT_AD.B,AX
2979 66| 58                    POP     EAX
297B 66| C1 E0 04              SHL     EAX,4
297F BB 08D0 R                  MOV     BX,OFFSET GDT
2982 66| 03 C3                 ADD     EAX,EBX
2985 2E: A3 2909 R              MOV     GDT_AD.A,AX
2989 66| C1 E8 10              SHR     EAX,16
298D 2E: A3 290D R              MOV     GDT_AD.CC,AX            ;set up GDT address
2991 2E: 0F 01 16 2909 R        LGDT    GDT_AD                  ;load GDT register
2997 0F 20 C0                   MOV     EAX,CR0                 ;set PM
299A 66| 83 C8 01              OR      EAX,1
299E 0F 22 C0                   MOV     CR0,EAX                 ;protected mode
29A1 EB 00                      JMP     NEXT

29A3                    NEXT:

29A3 B8 0008                    MOV     AX,8
29A6 0F 00 D8                   LTR     AX                      ;address TSS1
29A9 B8 0010                    MOV     AX,10H
29AC FF E0                      JMP     AX                      ;jump to TSS2

29AE                    MAIN    ENDP
```

```
29AE                    DO_IDT   PROC   NEAR
                        ;set up interrupts (not shown here)
29AE                    DO_IDT   ENDP
29AE                    TASK1:                          ;main program starts here
29AE                    CODE     ENDS
                                 END    MAIN
```

## 9–5 VIRTUAL 8086 MODE

One special mode of operation not discussed thus far is the virtual 8086 mode. This special mode is designed so multiple 8086 real-mode software applications can execute at one time. Figure 9–21 illustrates two 8086 applications mapped into the 80386 using the virtual mode. If the operating system allows multiple applications to execute, it is usually done through a technique called *time slicing*. The operating system allocates a set amount of time to each task. For example, if three tasks are executing, the operating system can allocate 1 ms to each task. This means that after each millisecond, a task switch occurs to the next task. In this manner, all tasks receive a portion of the 80386 execution time, resulting in a system that appears to execute more than one task at a time. The task times can be adjusted to give any task any percentage of the microprocessor execution time.

A system that can use this technique is a print spooler. The print spooler can function in one DOS partition and be accessed 10 percent of the time. This allows the system to print using the print spooler, but it doesn't detract from the system because it only uses 10 percent of the system time.

The main difference between 80386 protected-mode operation and the virtual 8086 mode is the way the segment registers are interpreted by the microprocessor. In the virtual 8086 mode, the segment registers are used as they are in the real mode; that is, as a segment address and offset address capable of accessing a 1M byte memory space from locations 00000H–FFFFFH. Access to many virtual 8086 mode systems is made possible by the paging unit explained in the next section. Through paging, the program accesses memory below the 1M byte boundary, but the microprocessor can access physical memory space at any location in the 4G-byte range of the memory system.

Virtual 8086 mode is entered by changing the VM bit in the EFLAG register to a logic 1. If the privilege level is 00, this mode is entered via an IRET instruction. This bit cannot be set in any other manner. An attempt to access a memory address above the 1M byte boundary will cause a type 13 interrupt to occur.

The virtual 8086 mode could be used to share one microprocessor among many users by partitioning the memory so each user has his or her own DOS partition. User 1 could be allocated memory locations 00100000H–001FFFFFH, user 2 could be allocated memory locations 0020000H–002FFFFFH, and so forth. The system software located at memory locations 00000000H–000FFFFFH could then share the microprocessor between users by switching from one memory location to another to execute software.

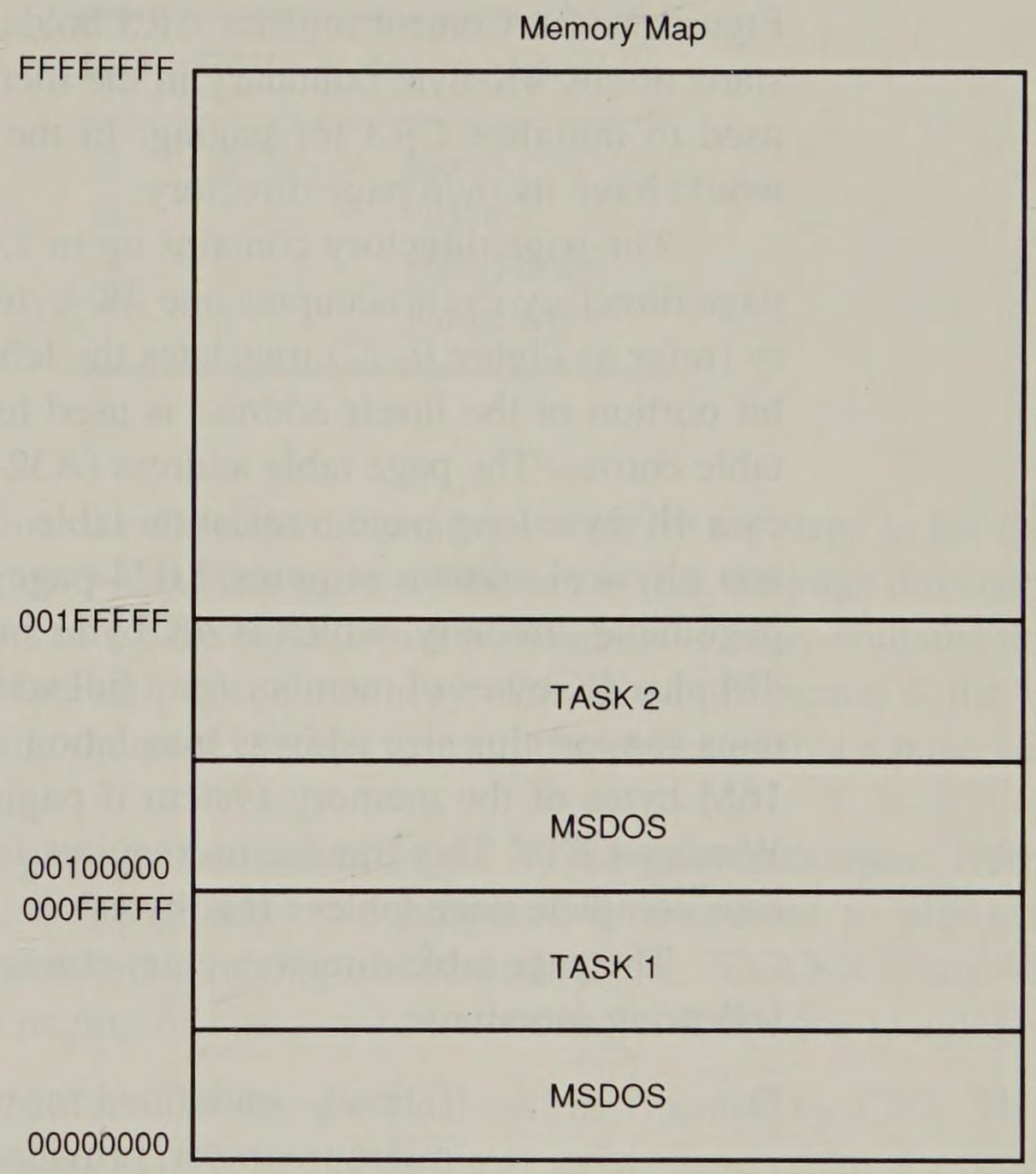



**FIGURE 9–21** Two tasks resident to an 80386 operated in the virtual 8086 mode.

# 9–6 THE MEMORY PAGING MECHANISM

The paging mechanism allows any linear (logical) address, as it is generated by a program, to be placed into any physical memory page, as generated by the paging mechanism. A *linear memory page* is a page that is addressed with a selector and an offset in either the real or protected mode. A *physical memory page* is a page that exists at some actual physical memory location. For example, linear memory location 20000H could be mapped into physical memory location 30000H, or any other location, with the paging unit. This means that an instruction that accesses location 20000H actually accesses location 30000H.

Each 80386 memory page is 4K bytes in length. Paging allows the system software to be placed at any physical address with the paging mechanism. Three components are used in page address translation: the page directory, the page table, and the actual physical memory page. Note that EEM386.EXE, the extended memory manager, uses the paging mechanism to simulate expanded memory in extended memory and also to generate upper memory blocks between system ROMs.

## The Page Directory

The page directory contains the location of up to 1,024 page translation tables. Each page translation table translates a logic address into a physical address. The page directory is stored in the memory and accessed by the page descriptor address register (CR3) (see

Figure 9–10). Control register CR3 holds the base address of the page directory, which starts at any 4K-byte boundary in the memory system. The MOV CR3,reg instruction is used to initialize CR3 for paging. In the virtual 8086 mode, each 8086 DOS partition would have its own page directory.

The page directory contains up to 1,024 entries that are each 4 bytes in length. The page directory itself occupies one 4K-byte memory page. Each entry in the page directory (refer to Figure 9–22) translates the left-most 10 bits of the memory address. This 10-bit portion of the linear address is used to locate different page tables for different page table entries. The page table address (A32–A12), stored in a page directory entry, accesses a 4K byte long page translation table. To completely translate any linear address into any physical address requires 1,024 page tables that are each 4K bytes in length plus a page table directory, which is 4K bytes in length. This translation scheme requires up to 4M plus 4K bytes of memory for a full address translation. Only the largest operating systems support this size address translation. Many operating systems translate only the first 16M bytes of the memory system if paging is enabled. This includes programs such as Windows 3.1.* This translation requires four entries in the page directory (16 bytes) and four complete page tables (16K bytes).

The page table directory entry control bits, as illustrated in Figure 9–22, perform the following functions:

D — (Dirty)—undefined for page table directory entries by the 80386 microprocessor, provided for use by the operating system.

A — (Accessed)—is set to a logic 1 whenever the microprocessor accesses the page directory entry.

R/W and U/S — (Read/Write and User/Supervisor)—used in the protection scheme as listed in Table 9–3. Both bits combine to develop paging priority level protection for level 3, the lowest user level.

P — (Present)—indicates that the entry can be used in address translation if P = 1. If P = 0, the entry cannot be used for translation. A not present entry can be used for other purposes, such as indicating that the page is currently stored on the disk. If P = 0, the remaining bits of the entry can be used to indicate the location of the page on the disk memory system.

## Page Table

The page table contains 1,024 physical page addresses accessed to translate a linear address into a physical address. Each page table translates a 4M section of the linear memory into

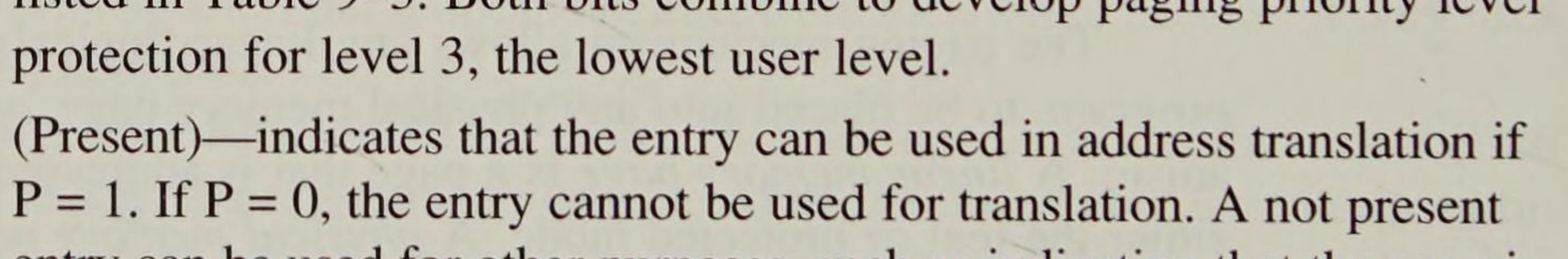



**FIGURE 9–22** The page table directory entry.

*Windows is a program produced by Microsoft Corporation.

**TABLE 9–3** Protection for level 3 using U/S and R/W

| *U/S* | *R/W* | *Access Level 3* |
|---|---|---|
| 0 | 0 | None |
| 0 | 1 | None |
| 1 | 0 | Read-only |
| 1 | 1 | Read/write |

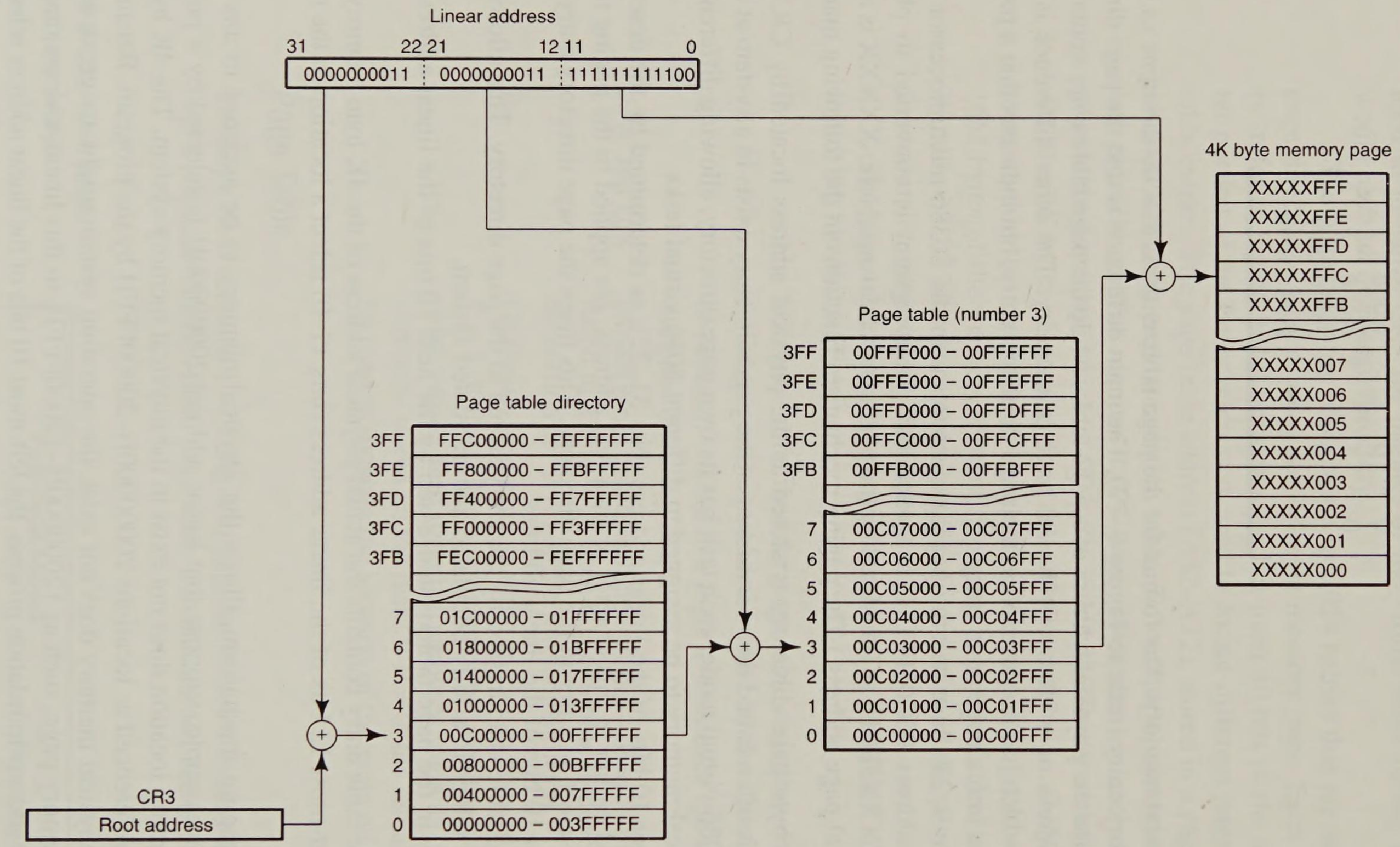
4M of physical memory. The format for the page table entry is exactly the same as for the page directory entry (refer to Figure 9–22). The main difference is that the page directory entry contains the physical address of a page table, while the page table entry contains the physical address of a 4K byte physical page of memory. The other difference is the D (dirty bit), which has no function in the page directory entry, but indicates that a page has been written into a page table entry.

Figure 9–23 illustrates the paging mechanism in the 80386 microprocessor. Here, the linear address 00C03FFCH, as generated by a program, is converted to physical address XXXXX3FCH, as translated by the paging mechanism. (Note: XXXXX is any 4K byte physical page address.) The paging mechanism functions in the following manner:

1. The 4K byte page directory is stored as the physical address located by CR3. This address is often called the *root address*. One page directory exists in a system at a time. In the 8086 virtual mode, each task has its own page directory, allowing different areas of physical memory to be assigned to different 8086 virtual tasks.
2. The upper 10 bits of the linear address (bits 31–22), as determined by the descriptors described earlier in this chapter or by a real address, are applied to the paging mechanism to select an entry in the page directory. This maps the page directory entry to the left-most 10 bits of the linear address.
3. The page table is addressed by the entry stored in the page directory. This allows up to 4K page tables in a fully populated and translated system.
4. An entry in the page table is addressed by the next 10 bits of the linear address (bits 21–12).
5. The page table entry contains the actual physical address of the 4K byte memory page.
6. The right-most 12 bits of the linear address (bits 11–0) select a location in the memory page.

The paging mechanism allows the physical memory to be assigned to any linear address. For example, suppose that linear address 20000000H is selected by a program, but this memory location does not exist in the physical memory system. The 4K byte linear page is referenced as locations 20000000H–20000FFFH by the program. Because this section of physical memory does not exist, the operating system might assign an existing physical memory page, such as 12000000H–12000FFFH, to this linear address range.

In the address translation process, the left-most 10 bits of the linear address select page directory entry 200H, located at offset address 800H in the page directory. This page directory entry contains the address of the page table for linear addresses 20000000H–203FFFFFH. Linear address bits 21–12 select an entry in this page table that corresponds to a 4K byte memory page. For linear addresses 2000000H–20000FFFH, the first entry (entry 0) in the page




Note: 1. The address ranges illustrated in the page directory and page table represent the linear address ranges selected and not the contents of these tables.

2. The addresses (XXXXX) listed in the memory page are selected by the page table entry.

**FIGURE 9–23** The translation of linear address 00CO3FFC to physical memory address XXXXXFFC. The value of XXXXX is determined by the page table entry (not shown here).

table is selected. This first entry contains the physical address of the actual memory page, or 12000000H–12000FFFH in this example.

Take, for example, a typical DOS-based computer system. The memory map for the system appears in Figure 9–24. Notice from the map that there are unused areas of memory that could be paged to a different location, giving a DOS real-mode application program more memory. The normal DOS memory system begins at location 00000H and extends to location 9FFFFH, which is 640K bytes of memory. Above location 9FFFFH, we find sections devoted to video cards, disk cards, and the system BIOS ROM. In this example, an area of memory just above 9FFFFH is unused (A0000–AFFFFH). This section of the memory could be used by DOS so that the total applications memory area is 704K instead of 640K. Be careful when using A0000H–AFFFFH for additional RAM because the video card uses this area for bit-mapped graphics in modes 12H and 13H.

This section of memory can be used by mapping it into extended memory at locations 102000H–11FFFFH. Software to accomplish this translation and initialize the page table directory and page tables required to set up memory are illustrated in Example 9–3. Note that this procedure initializes the page table directory, a page table, and loads CR3.

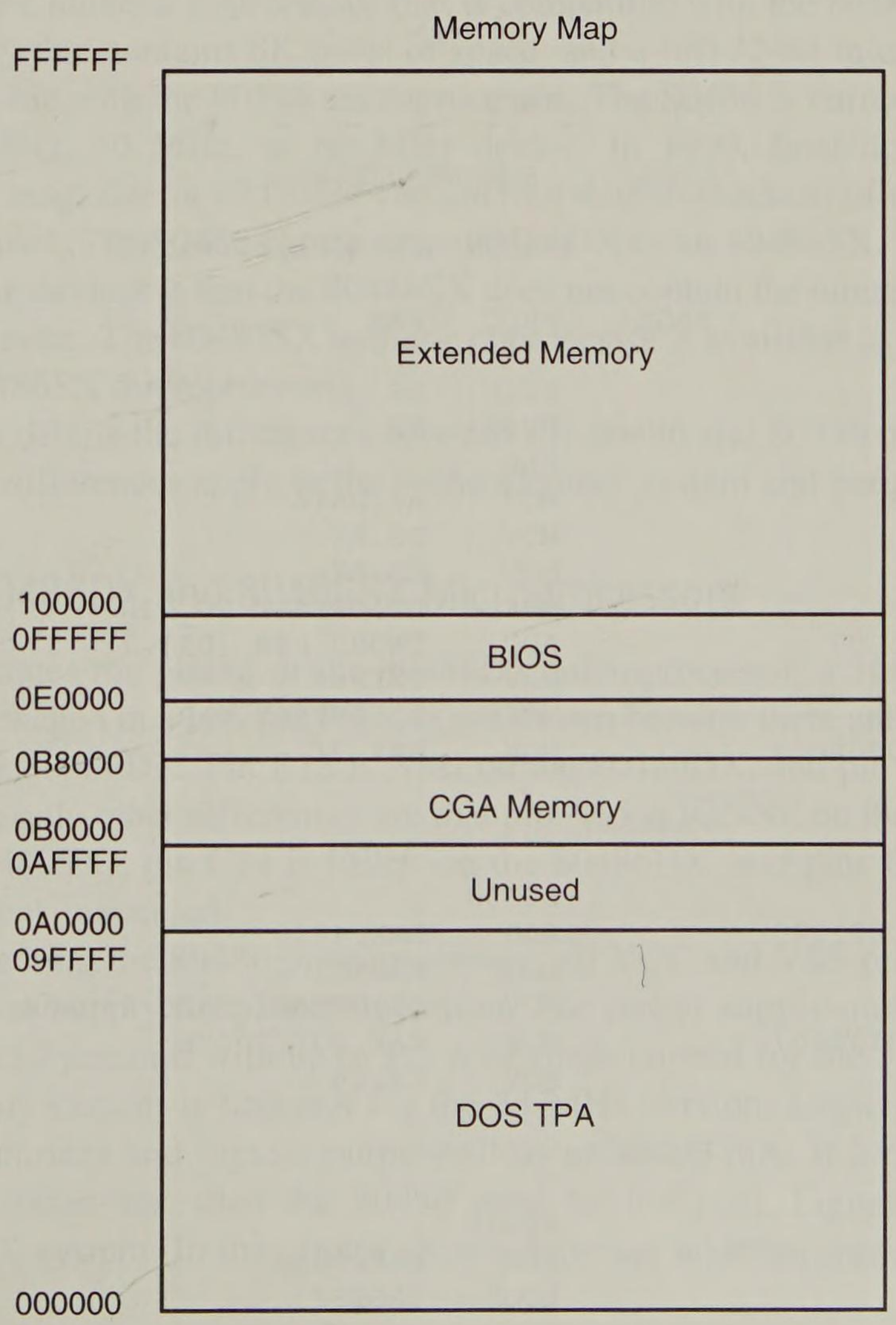



**FIGURE 9–24** Memory map for an AT-style clone.

It does not switch to protected mode; it does enable paging. Note that paging functions in real-mode memory operation.

**EXAMPLE 9–3**

```
                        .386P
                        ;Software that sets up a paged memory system
                        ;that accesses memory above location FFFFFH.
                        ;
                        ;Page Directory
0000                    DATA    SEGMENT

0000 = 0000             PG_DIR  EQU   $

0000 00001000 R         TAB0PT  DD    TAB0
0004 03FF [                     DD    1023 DUP (?)
           00000000
                   ]

                        ;Page Table

1000 0400 [             TAB0    DD     1024 DUP (?)
           00000000
                   ]

2000                    DATA    ENDS

0000                    CODE    SEGMENT USE16

                                ASSUME CS:CODE,DS:DATA

0000                    PAGES   PROC   FAR

0000 00 1E                      PUSH   DS
0001 06                         PUSH   ES
0002 FC                         CLD
0003 B8 ---- R                  MOV    AX,DATA                  ;load segment registers
0006 8E D8                      MOV    DS,AX
0008 8E C0                      MOV    ES,AX
000A BF 0000 R                  MOV    DI,OFFSET PG_DIR         ;address page table
000D 66| 83 05 07               ADD    DWORD PTR [DI],7
0011 B9 0100                    MOV    CX,256
0014 BF 1000 R                  MOV    DI,OFFSET TAB0
0017 66| B8 00000007            MOV    EAX,7

001D                    PAGES1:

001D 66| AB                     STOSD                           ;fill page table
001F 66| 05 00001000           ADD    EAX,4096
0025 E2 F6                      LOOP   PAGES1
0027 BF 1280 R                  MOV    DI,OFFSET TAB0+4*0A0H
002A 66| B8 00102007            MOV    EAX,00102007H
0030 B9 0010                    MOV    CX,16

0033                    PAGES2:

0033 66| AB                     STOSD                           ;remap A0000H-AFFFFH
0035 66| 05 00001000           ADD    EAX,4096                 ;to 102000H-11FFFFH
003B E2 F6                      LOOP   PAGES2
```

```
003D 66| 33 C0                  XOR     EAX,EAX
0040 8C D8                      MOV     AX,DS
0042 66| C1 E0 04               SHL     EAX,4
0046 0F 22 D8                   MOV     C3,EAX                  ;address page directory

0049 07                         POP     ES
004A 1F                         POP     DS
004B CB                         RET

004C                    PAGES   ENDP

004C                    CODE    ENDS

                        END
```

## 9–7 INTRODUCTION TO THE 80486 MICROPROCESSOR

The 80486 microprocessor is a highly integrated device containing well over 1.2 million transistors. Located within this powerful integrated circuit are a memory management unit (MMU); a complete numeric coprocessor that is compatible with the 80387; a high-speed one-cache memory that contains 8K bytes of space; and a full 32-bit microprocessor that is upward compatible with the 80386 microprocessor. The 80486 is currently available as a 25 MHz, 33 MHz, 50 MHz, or 66 MHz device. In 1990, Intel demonstrated, for Computer Design magazine, a 100 MHz version (not double-clocked) of the 80486, but it has yet to be released. The 80486 comes as an 80486DX or an 80486SX. The only difference between these devices is that the 80486SX does not contain the numeric coprocessor, which reduces its price. The 80487SX numeric coprocessor is available as a separate component for the 80486SX microprocessor.

This section details the differences between the 80486 and 80386 microprocessors. The most notable differences apply to the cache memory system and parity generator.

### Pinout of the 80486DX and 80486SX Microprocessors

Figure 9–25 illustrates the pinout of the 80486DX microprocessor, a 168-pin PGA. The 80486SX, also packaged in a 168-pin PGA, is not shown because there are few differences between it and the 80486DX. Pin B15 is NMI on the 80486DX, and pin A15 is NMI on the 80486SX. The only other differences are that pin A15 is IGNNE on the 80486DX (not present on the 80486SX), pin C14 is $\overline{\text{FERR}}$ on the 80486DX, and pins B15 and C14 on the 80486SX are not connected.

When connecting the 80486 microprocessor, all VCC and VSS pins must be connected to the power supply for proper operation. The power supply must be capable of supplying 5.0 V ±10 percent, with up to 1.2 A of surge current for the 33 MHz version. The average supply current is 650 mA for the 33 MHz version. Logic 0 outputs allow up to 4.0 mA of current and logic 1 outputs allow up to 1.0 mA. If larger currents are required, as they often are, then the 80486 must be buffered. Figure 9–26 shows a buffered 80486DX system. In the circuit shown, only the address, data, and parity signals are buffered.

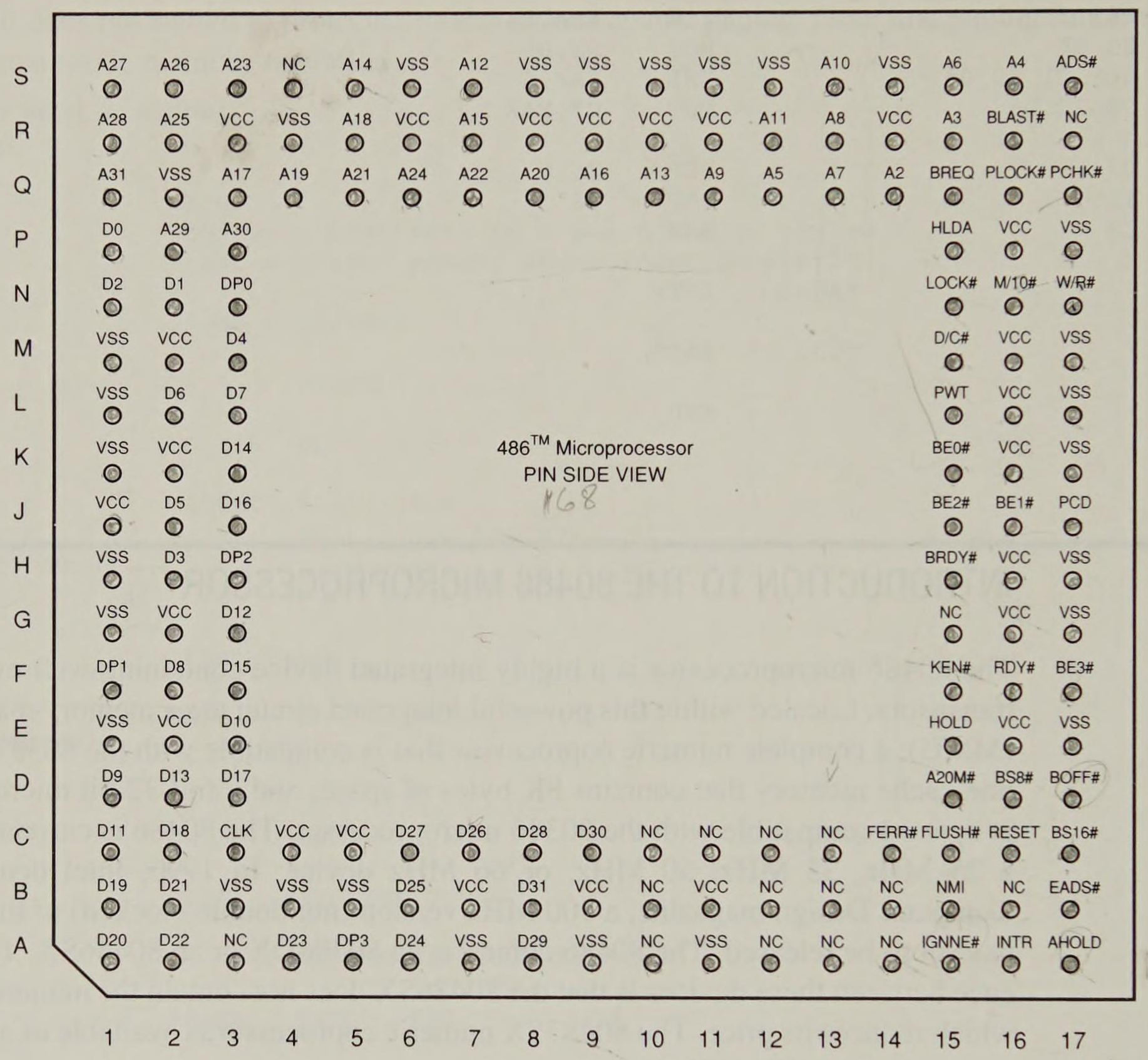



**FIGURE 9–25** The pinout of the 80486. (Courtesy of Intel Corporation)

*Pin Definitions.*

A31–A2 (Address Outputs)—provide the memory and I/O with the address during normal operation. During a cache line invalidation, A31–A4 are used to drive the microprocessor.

$\overline{A20M}$ (Address Bit 20 Mask)—used to cause the 80486 to wrap its address around from location 000FFFFFH to 00000000H, as in the 8086 microprocessor. This provides a memory system that functions like the 1M byte real memory system in the 8086 microprocessor.

$\overline{ADS}$ (Address Data Strobe)—becomes a logic zero to indicate that the address bus contains a valid memory address.

AHOLD (Address Hold Input)—causes the microprocessor to place its address bus connections at their high-impedance state, with the remainder of the buses staying active. Often used by another bus master to gain access for a cache invalidation cycle.

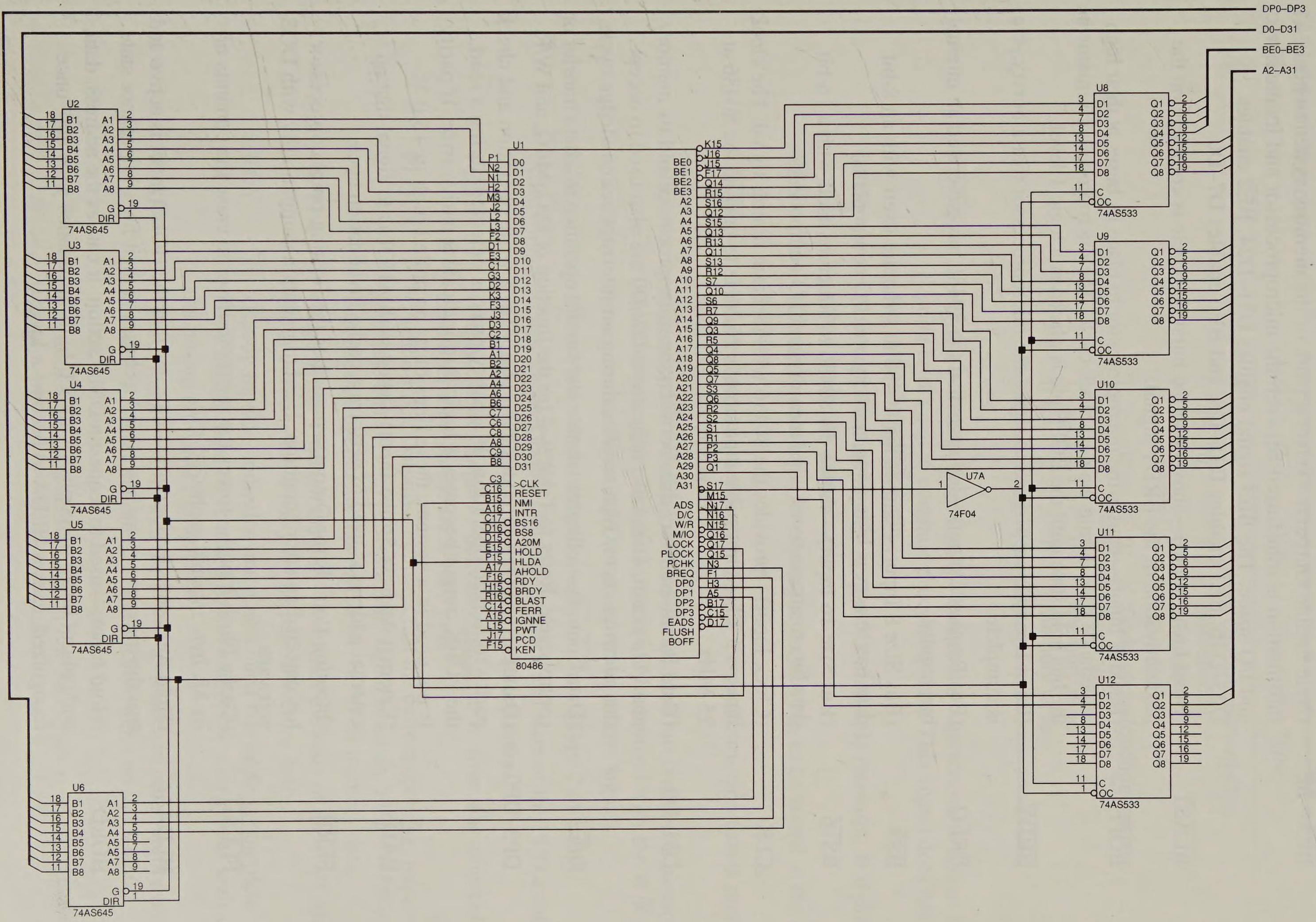



**FIGURE 9–26** An 80486 microprocessor showing the buffered address, data, and parity buses.

| | |
|---|---|
| $\overline{BE3}$—$\overline{BE1}$ | (Byte Enable Outputs)—select a bank of the memory system when information is transferred between the microprocessor and its memory and I/O space. The $\overline{BE3}$ signal enables D31–D24, $\overline{BE2}$ enables D23–D16, $\overline{BE1}$ enables D15–D8, and $\overline{BE0}$ enables D7–D0. |
| $\overline{BLAST}$ | (Burst Last Output)—shows that the burst bus cycle is complete on the next activation of the $\overline{BRDY}$ signal. |
| $\overline{BOFF}$ | (Backoff Input)—causes the microprocessor to place its buses at their high-impedance state during the next clock cycle. The microprocessor remains in the bus hold state until the BOFF pin is placed at a logic 1 level. |
| $\overline{BRDY}$ | (Burst Ready Input)—used to signal the microprocessor that a burst cycle is complete. |
| BREQ | (Bus Request Output)—indicates that the 80486 has generated an internal bus request. |
| $\overline{BS8}$ | (Bus Size 8 Input)—causes the 80486 to structure itself with an 8-bit data bus to access byte-wide memory and I/O components. |
| $\overline{BS16}$ | (Bus Size 16 Input)—causes the 80486 to structure itself with a 16-bit data bus to access word-wide memory and I/O components. |
| CLK | (Clock Input)—provides the 80486 with its basic timing signal. The clock input is a TTL compatible input that is 25 MHz to operate the 80486 at 25 MHz. |
| D31–D0 | (Data Bus)—transfers data between the microprocessor and its memory and I/O system. Data bus connections D7–D0 are also used to accept the interrupt vector type number during an interrupt acknowledge cycle. |
| D/$\overline{C}$ | (Data/$\overline{Control}$)—indicates whether the current operation is a data transfer or control cycle. Refer to Table 9–4 for the function of D/$\overline{C}$, M/$\overline{IO}$, and W/$\overline{R}$. |
| DP3–DP0 | (Data Parity I/O)—provides even parity for a write operation and check parity for a read operation. If a parity error is detected during a read, the $\overline{PCHK}$ output becomes a logic 0 to indicate a parity error. If parity is not used in a system, these lines must be pulled high to +5.0 V. |
| $\overline{EADS}$ | (External Address Strobe Input)—used with AHOLD to signal that an external address is used to perform a cache invalidation cycle. |
| $\overline{FERR}$ | (Floating-Point Error Output)—indicates the floating-point coprocessor has detected an error condition. Used to maintain compatibility with DOS software. |
| $\overline{FLUSH}$ | (Cache Flush Input)—forces the microprocessor to erase the contents of its 8K byte internal cache. |
| HLDA | (Hold Acknowledge Output)—indicates that the HOLD input is active and that the microprocessor has placed its buses at their high-impedance state. |
| HOLD | (Hold Input)—used to request a DMA action. It causes the address, data, and control buses to be placed at their high-impedance state and, once recognized, causes HLDA to become a logic 0. |

**TABLE 9–4** Bus cycle identification

| $M/\overline{IO}$ | $D/\overline{C}$ | $W/\overline{R}$ | *Bus Cycle Type* |
|---|---|---|---|
| 0 | 0 | 0 | Interrupt acknowledge |
| 0 | 0 | 1 | Halt/special |
| 0 | 1 | 0 | I/O read |
| 0 | 1 | 1 | I/O write |
| 1 | 0 | 0 | Code cycle (opcode fetch) |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Memory read |
| 1 | 1 | 1 | Memory write |

$\overline{IGNNE}$ (Ignore Numeric Error Input)—causes the coprocessor to ignore floating-point errors and to continue processing data. This signal does not affect the state of the $\overline{FERR}$ pin.

INTR (Interrupt Request Input)—requests a maskable interrupt, as it does on all other family members.

$\overline{KEN}$ (Cache Enable Input)—causes the current bus to be stored in the internal cache.

$\overline{LOCK}$ (Lock Output)—becomes a logic 0 for any instruction that is prefixed with the lock prefix.

$M/\overline{IO}$ (Memory/$\overline{IO}$)—defines whether the address bus contains a memory address or an I/O port number. It is also combined with the $W/\overline{R}$ signal to generate memory and I/O read and write control signals.

NMI (Non-Maskable Interrupt Input)—requests a type 2 interrupt.

PCD (Page Cache Disable Output)—reflects the state of the PCD attribute bit in the page table entry or the page directory entry.

$\overline{PCHK}$ (Parity Check Output)—indicates that a parity error was detected during a read operation on the DP3–DP0 pins.

$\overline{PLOCK}$ (Pseudo-Lock Output)—indicates that the current operation requires more than one bus cycle to perform. This signal becomes a logic 0 for arithmetic coprocessor operations that access 64- or 80-bit memory data.

PWT (Page Write Through Output)—indicates the state of the PWT attribute bit in the page table entry or the page directory entry.

$\overline{RDY}$ (Ready Input)—indicates that a non-burst bus cycle is complete. The $\overline{RDY}$ signal must be returned or the microprocessor places wait states into its timing until $\overline{RDY}$ is asserted.

RESET (Reset Input)—initializes the 80486 as it does in other family members. Table 9–5 shows the effect of the RESET input on the 80486 microprocessor.

$W/\overline{R}$ (Write/$\overline{Read}$)—signals that the current bus cycle is either a read or a write.

**TABLE 9–5** The effect of the RESET signal

| *Register* | *Initial Value With Self Test* | *Initial Value Without Self Test* |
|---|---|---|
| EAX | 00000000H | ? |
| EDX | 00000400H + ID* | 00000400H + ID |
| EFLAGS | 00000002H | 00000002H |
| EIP | 0000FFF0H | 0000FFF0H |
| ES | 0000H | 0000H |
| CS | F000H | F000H |
| DS | 0000H | 0000H |
| SS | 0000H | 0000H |
| FS | 0000H | 0000H |
| GS | 0000H | 0000H |
| IDTR | Base = 0, limit 3FFH | Base = 0, limit = 3FFH |
| CR0 | 60000010H | 60000010H |
| DR7 | 00000000H | 00000000H |

**Note*: Revision ID number supplied by Intel for revisions to the microprocessor.

## Basic 80486 Architecture

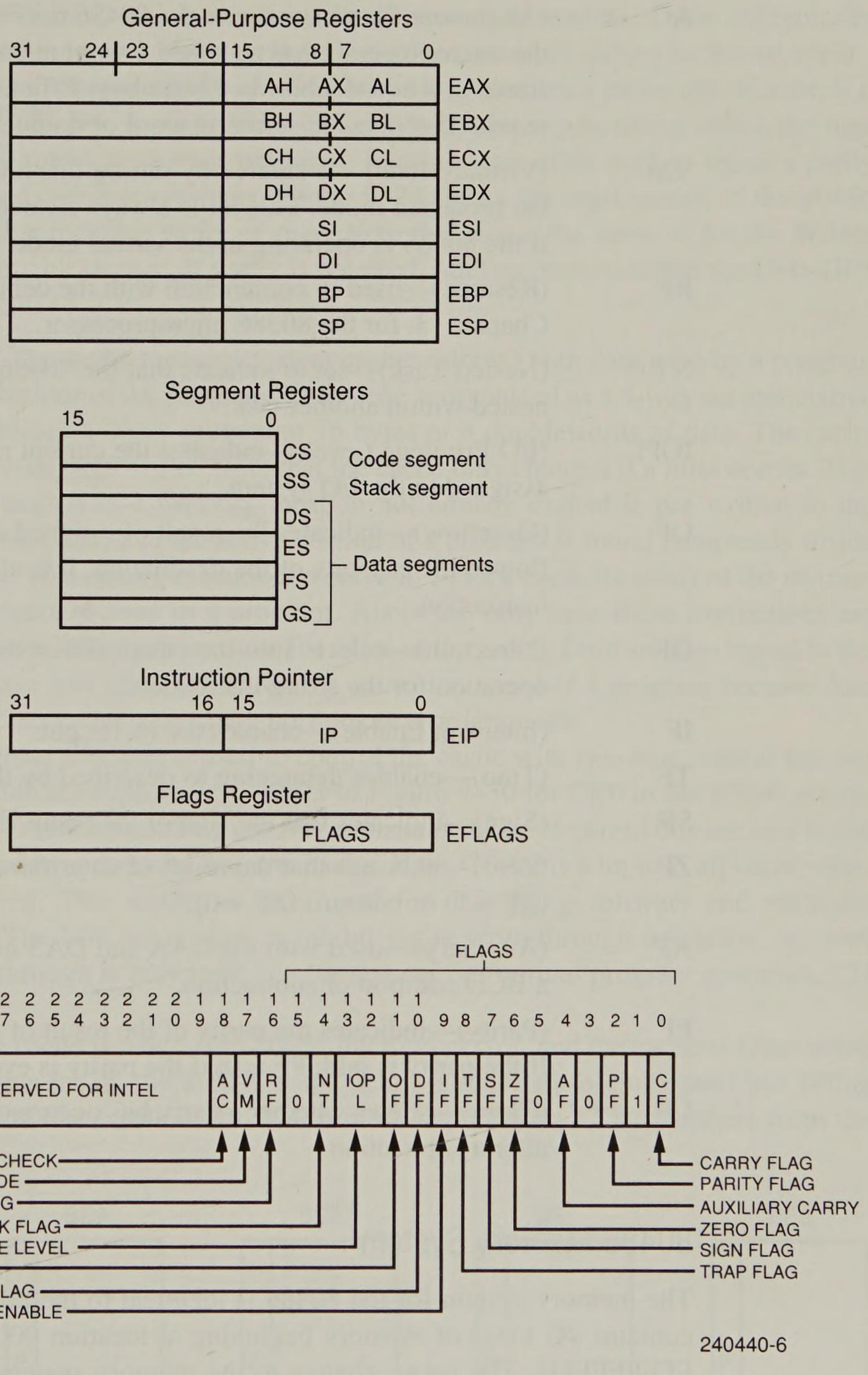
The architecture of the 80486DX is almost identical to the 80386. Added to the 80386 architecture inside the 80486DX is a math coprocessor and an 8K byte one-cache memory. The 80486SX is almost identical to an 80386 with an 8K byte cache, but no numeric coprocessor. Figure 9–27 illustrates the basic internal structure of the 80486 microprocessor. If this is compared to the architecture of the 80386, no significant differences are observed. The most prominent difference between the 80386 and the 80486 is that almost half of the 80486 instructions execute in 1 clocking period, rather than the 2 clocking periods required by the 80386 to execute like instructions.

Like the 80386, the 80486 contains eight general purpose 32-bit registers: EAX, EBX, ECX, EDX, EBP, EDI, ESI, and ESP. These registers may be used as 8-, 16-, or 32-bit data registers, or to address a location in the memory system. The 16-bit registers are the same set as found in the 80286 and are assigned: AX, BX, CX, DX, BP, DI, SI, and SP. The 8-bit registers are: AH, AL, BH, BL, CH, CL, DH, and DL.

In addition to the general purpose registers, the 80486 also contains the same segment registers as the 80386, which are: CS, DS, ES, SS, FS, and GS. Each is 16 bits wide, as in all earlier versions of the family.

The IP (instruction pointer) addresses the program located within the 1M byte of memory, in combination with CS or EIP (extended instruction pointer), to address a program at any location within the 4G byte memory system. In protected-mode operation, the segment registers function to hold selectors, as they did in the 80286 and 80386 microprocessors.

The 80486 also contains the same global, local, and interrupt descriptor table register and memory management unit as the 80386. Although these registers are not illustrated in Figure 9–27, they are present just as they are in the 80386. The function of the MMU and its paging unit is described earlier in the chapter.




NOTE:
0 indicates Intel Reserved: do not define; see Section 2.1.6.

**FIGURE 9–28** The EFLAG register of the 80486. (Courtesy of Intel Corporation)

The extended flag register (EFLAGS) is illustrated in Figure 9–28. The right-most flag bits perform the same functions for compatibility that they do with other family members. Following is a list of each flag bit with a description of its function:

AC (Alignment Check)—new to the 80486 microprocessor, used to indicate that the microprocessor has accessed a word at an odd address or a doubleword stored at a non-doubleword boundary. Efficient software and execution require that data are stored at word or doubleword boundaries.

VM (Virtual Mode)—is entered by setting this bit while the 80486 is operated in the protected mode. This bit is always cleared by a PUSHF instruction, even if the 80486 is operating in the virtual mode.

RF (Resume)—used in conjunction with the debug registers, as defined in Chapter 13, for the 80386 microprocessor.

NT (Nested Task)—set to indicate that the 80486 is performing a task that is nested within another task.

IOPL (I/O Privilege Level)—indicates the current maximum privilege level assigned to the I/O system.

OF (Overflow)—indicates the result of a signed arithmetic operation has overflowed the capacity of the destination. It is also used with the multiply instruction.

DF (Direction)—selects auto-increment (DF = 0) or auto-decrement (DF = 1) operation for the string instructions.

IF (Interrupt Enable)—enables the INTR pin.

TF (Trap)—enables debugging as described by the debug registers.

SF (Sign)—indicates that the sign of the result is set or cleared.

ZF (Zero)—indicates that the result of an arithmetic or logic operation is zero (ZF = 1) or nonzero (ZF = 0).

AF (Auxiliary)—used with the DAA and DAS instructions to adjust the result of a BCD addition or subtraction.

PF (Parity)—indicates the parity of the result of an arithmetic or logic operation. If the parity is odd, PF = 0; if the parity is even, PF = 1.

CF (Carry)—shows whether a carry has occurred after an addition or a borrow after a subtraction.

## 80486 Memory System

The memory system for the 80486 is identical to the 80386 microprocessor. The 80486 contains 4G bytes of memory beginning at location 00000000H and ending at location FFFFFFFFH. The major change to the memory system is internal to the 80486 in the form of an 8K byte cache memory, which speeds the execution of instructions and the acquisition of data. Another addition is the parity checker/generator built into the 80486 microprocessor.

*Parity Checker/Generator.* *Parity* is often used to determine whether data are read correctly from a memory location. To facilitate this, Intel has incorporated an internal *parity generator/detector*. Parity is generated by the 80486 during each write cycle. Parity is generated as *even parity* and a parity bit is provided for each byte of memory. The parity check bits

appear on pins DP0–DP3, which are parity inputs as well as outputs. These are typically stored in memory during each write cycle and read from memory during each read cycle.

On a read, the microprocessor checks parity and generates a parity check error, *if it occurs*, on the $\overline{\text{PCHK}}$ pin. A parity error causes no change in processing unless the user applies the $\overline{\text{PCHK}}$ signal to an interrupt input. Interrupts are often used to signal a parity error in DOS-based computer systems. Figure 9–29 shows the organization of the 80486 memory system that includes parity storage. Note that this is the same as for the 80386, except for the parity bit storage. If parity is not used, Intel recommends that the DP0–DP3 pins are pulled up to +5.0 V.

*Cache Memory.* The cache memory system caches (stores) both data used by a program and also the instructions of the program. The *cache* is organized as a 4-way set associative cache with each location (line) containing 16 bytes or 4 doublewords of data. The cache operates as a *write-through* cache. Note that the cache only changes if a miss occurs. This means that data written to a memory location not already cached is not written to the cache. In many cases, much of the active portion of a program is found completely inside the cache memory. This causes execution to occur in 1 clock cycle for many of the instructions that are commonly used in a program. About the only time these instructions are slowed is when the microprocessor must fill a line in the cache. Data are also stored in the cache, but they have less of an impact on the execution speed of a program because data are not referenced repeatedly, as many portions of a program are.

Control register 0 (CR0) is used to control the cache with two new control bits not present in the 80386 microprocessor. (Refer to Figure 9–30 for CR0 in the 80486 microprocessor.) The CD (*cache disable*) and NW (*noncache write-through*) bits are new to the 80486 and are used to control the 8K byte cache. If the CD bit is a logic 1, all cache operations are inhibited. This setting is only used for debugging software and normally remains cleared. The NW bit is used to inhibit cache write-through operation. As with CD, cache write-through is only inhibited for testing. For normal program operation, CD = 0 and NW = 0.

Because the cache is new to the 80486 microprocessor, and the cache is filled using burst cycles not present on the 80386, some detail is required to understand bus filling cycles. When a bus line is filled, the 80486 must acquire four 32-bit numbers from the

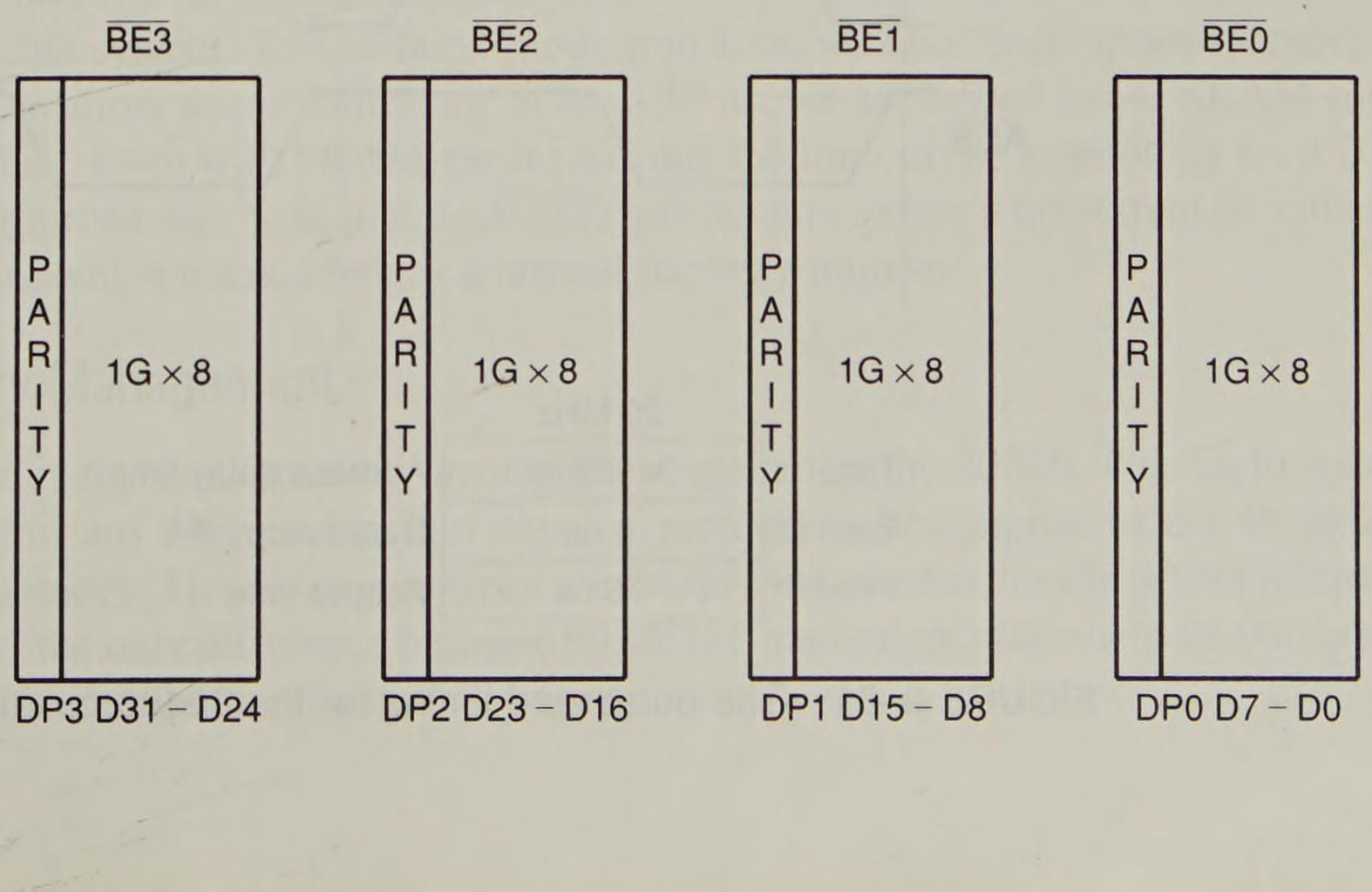



**FIGURE 9–29** The organization of the 80486 memory showing parity.

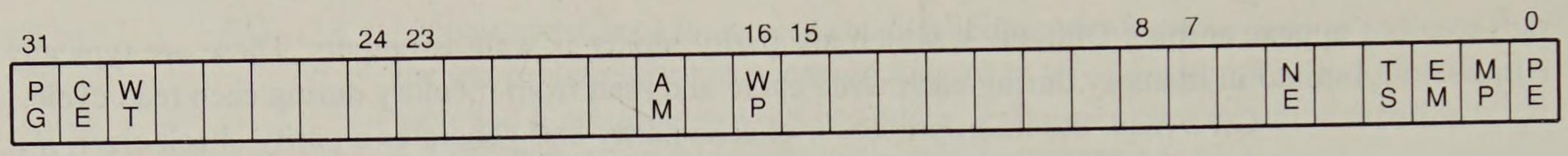
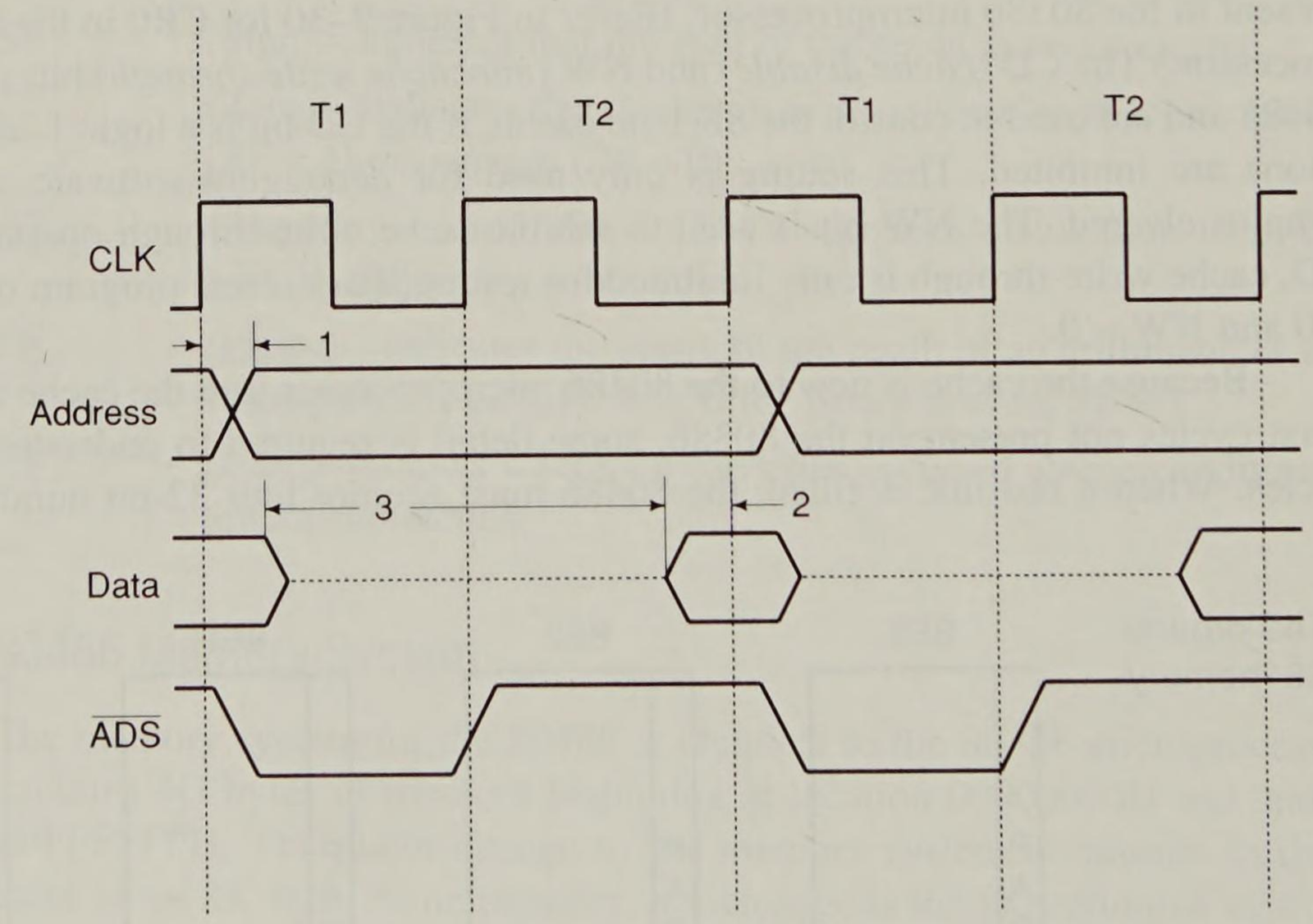



**FIGURE 9–30** Control register zero (CR0) for the 80486 microprocessor.

memory system to fill a line in the cache. Filling is accomplished with a *burst cycle*. The burst cycle is a special memory where four 32-bit numbers are fetched from the memory system in 5 clocking periods. This assumes that the speed of the memory is sufficient and that no wait states are required. If the clock frequency of the 80486 is 33 MHz we can fill a cache line in 167 ns, which is very efficient, considering a normal, non-burst 32-bit memory read operation requires 2 clocking periods.

*Memory Read Timing.* Figure 9–31 illustrates the read timing for the 80486 for a non-burst memory operation. Note that two clocking periods are used to transfer data. Clocking period T1 provides the memory address and control signals and clocking period T2 transfers data between the memory and the microprocessor. Note that the $\overline{\text{RDY}}$ must be logic zero to cause data to be transferred and to terminate the bus cycle. Access time for a non-burst access is determined by taking 2 clocking periods, minus the time required for the address to appear on the address bus connection, minus a setup time for the data bus con-




| | 20 MHz | |
|---|---|---|
| Time 1: | 3 – 28 ns | Address delay time |
| Time 2: | 6 ns | Data setup time |
| Time 3: | 76 ns | Access time |

**FIGURE 9–31** The burst read timing for the 80486 microprocessor.

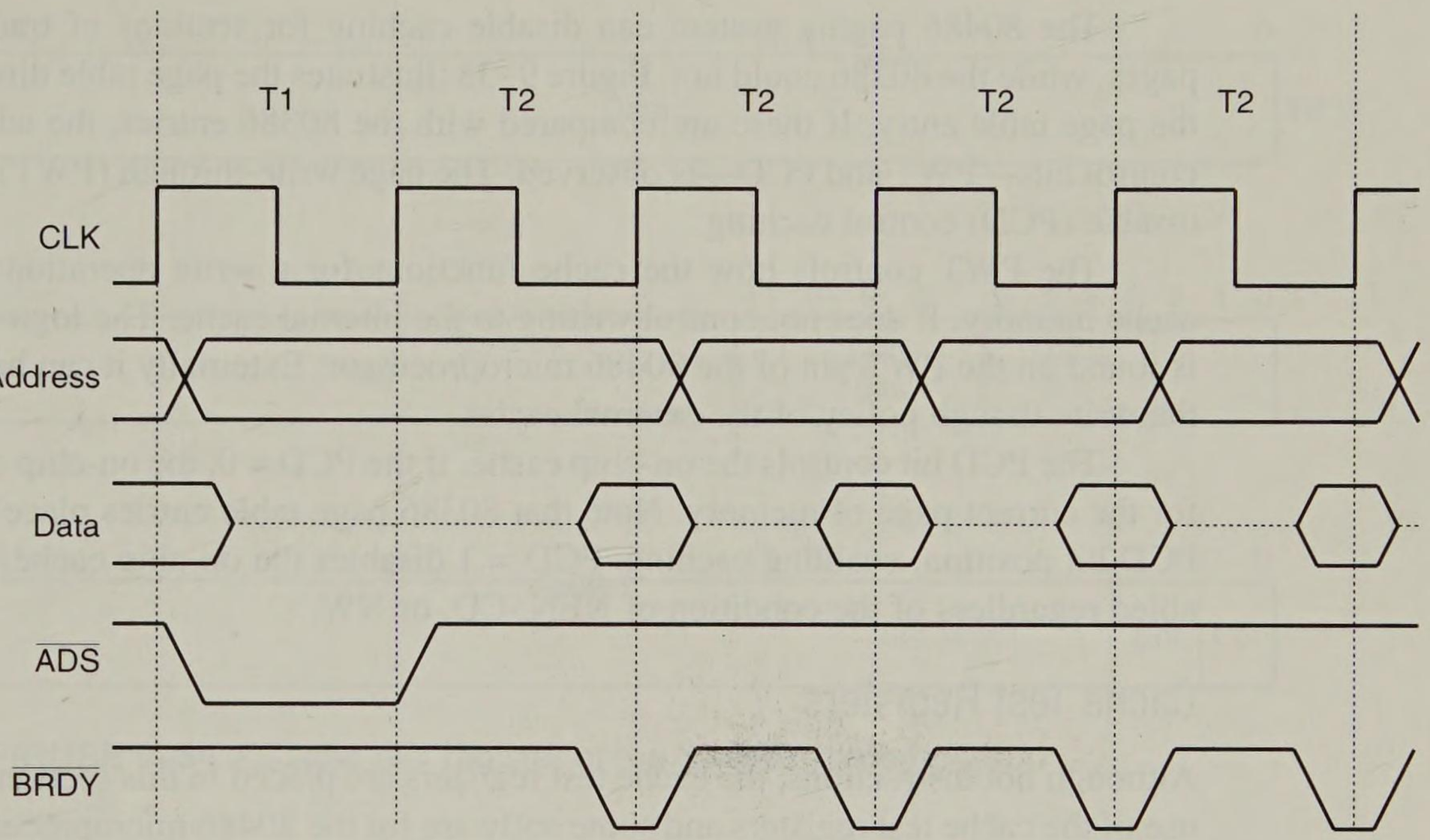



**FIGURE 9–32** A burst cycle that reads 4 doublewords in 5 clocking periods in the 80486 system.

nections. For the 20 MHz version of the 80486, two clocking periods require 100 ns, minus 28 ns for address setup time, minus 3 ns for data setup time. This yields a non-burst access time of 100 ns – 31 ns, or 69 ns. Of course, if decoder time and delay times are included, the access time allowed the memory is even less for no wait state operation. If a higher frequency version of the 80486 is used in a system, memory access time is even less.

Figure 9–32 illustrates the timing diagram for filling a cache line with four 32-bit numbers using a burst. Notice that the address (A31–A4) appears during T1 and remains constant throughout the burst cycle. Also notice that A2 and A3 change during each T2 after the first to address four consecutive 32-bit numbers in the memory system. As mentioned, cache fills using bursts require only five clocking periods (one T1 and 4 T2) to fill a cache line with four doublewords of data. Access time using a 20 MHz version of the 80486 for the second and subsequent doublewords is 50 ns – 28 ns – 3 ns, or 19 ns, assuming no delays in the system. To use burst mode transfers, we need high-speed memory. Because DRAM memory access times are, at best, 40 ns, we are forced to use SRAM for burst cycle transfers. Even with SRAM we are pushing the limit of the technology for a 20 MHz 80486 microprocessor. Note that the $\overline{\text{BRDY}}$ pin acknowledges a burst transfer rather than the $\overline{\text{RDY}}$ pin, which acknowledges a normal memory transfer.

## 80486 Memory Management

The 80486 contains the same memory management system as the 80386. This includes a paging unit to allow any 4K byte block of physical memory to be assigned to any 4K byte block of linear memory. The descriptor types are exactly the same as for the 80386 microprocessor. In fact, the only difference between the 80386 memory management system and the 80486 memory management system is paging.

The 80486 paging system can disable caching for sections of translated memory pages, while the 80386 could not. Figure 9–33 illustrates the page table directory entry and the page table entry. If these are compared with the 80386 entries, the addition of 2 new control bits—PWT and PCD—is observed. The page write-through (PWT) and page cache disable (PCD) control caching.

The PWT controls how the cache functions for a write operation of the external cache memory. It does not control writing to the internal cache. The logic level of this bit is found on the PWT pin of the 80486 microprocessor. Externally it can be used to dictate the write-though policy of the external cache.

The PCD bit controls the on-chip cache. If the PCD = 0, the on-chip cache is enabled for the current page of memory. Note that 80386 page table entries place a logic 0 in the PCD bit position, enabling caching. PCD = 1 disables the on-chip cache. Caching is disabled regardless of the condition of $\overline{\text{KEN}}$, CD, or NW.

## Cache Test Registers

Although not instructions, the cache test registers are placed in this section to illustrate the use of the cache test registers and some software for the 80486 microprocessor. The 80486 cache test registers are TR3 (cache data register), TR4 (cache status test register), and TR5 (cache control test register). These three registers, undefined for the 80386 microprocessor, are illustrated in Figure 9–34.

The cache data register (TR3) is used to access either the cache fill buffer for a write test operation or the cache read buffer for a cache read test operation. This register is a window into the 8K byte cache memory located within the 80486 and is used for testing the cache. In order to fill or read a cache line (128 bits wide) TR3 must be written or read four times.

The contents of the set select field in TR5 determine which internal cache line is written or read through TR3. The 7-bit test field selects one of the 128 different 16-byte cache lines. The entry select bits of TR5 select an entry in the set or the 32-bit location in the fill/read buffer. The control bits in TR5 enable the fill buffer or read buffer operation (00), perform a cache write (01) or cache read (10), or flush the cache (11).

The cache status register (TR4) holds the cache tag, LRU bits, and a valid bit. This register is loaded with the tag and valid bit before a cache write operation, and contains the tag, valid bit, LRU bits, and four valid bits on a cache test read.

The cache is tested each time the microprocessor is reset if the AHOLD pin is high for 2 clocks prior to the RESET pin going low. This causes the 80486 to completely test itself with a built-in self-test or BIST. The BIST uses TR3, TR4, and TR5 to completely test the internal cache. Its outcome is reported in register EAX. If EAX is a zero, the microprocessor, coprocessor, and cache have passed the self-test. The value of EAX can be tested after a reset to determine whether an error is detected. In most cases, we do not directly access the test registers unless we wish to perform our own tests on the cache or TLB.

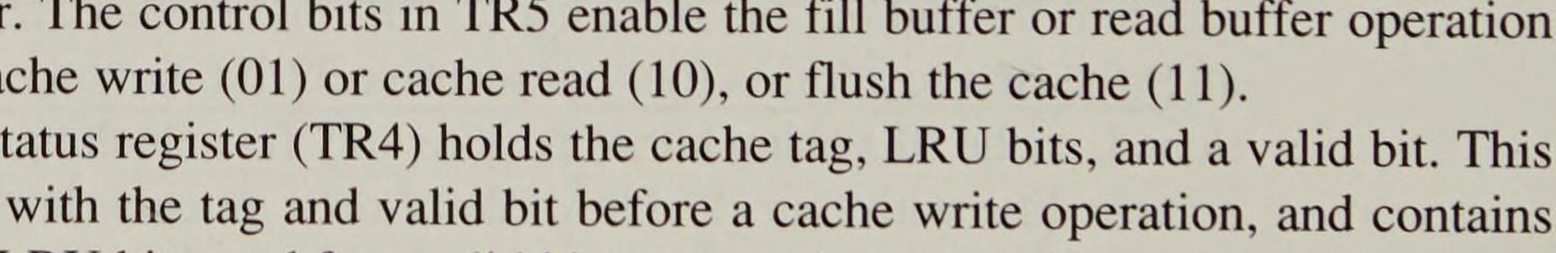



**FIGURE 9–33** The page directory or page table entry for the 80486 microprocessor.

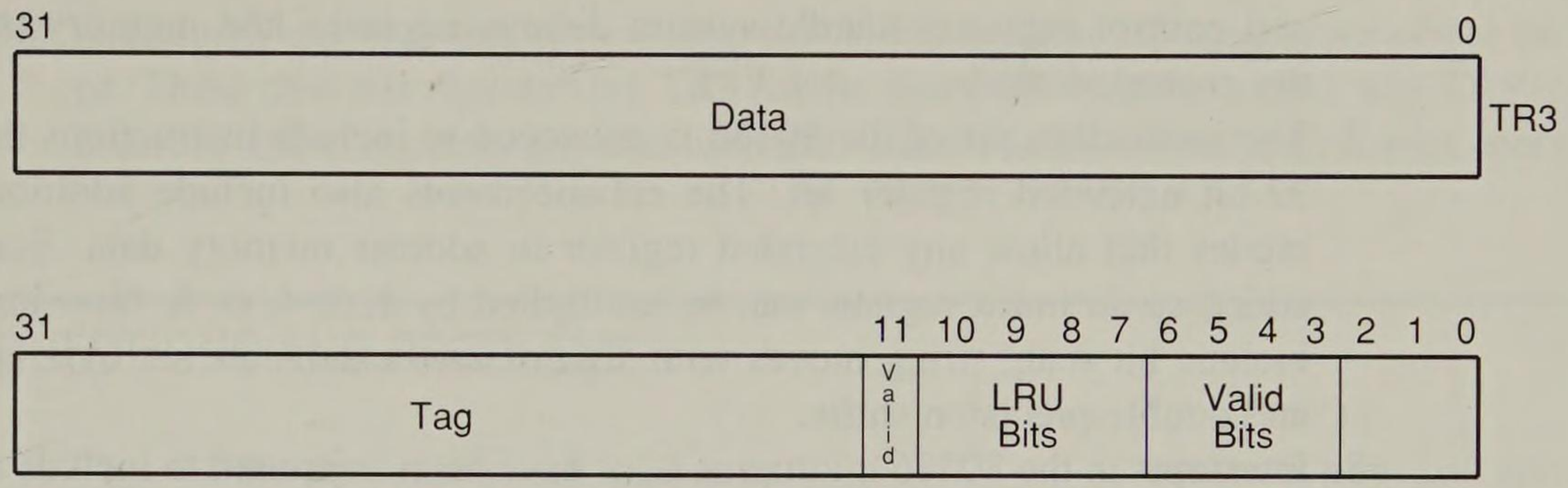



**FIGURE 9–34** Cache test register of the 80486 microprocessor.

## 9–8 SUMMARY

1. The 80386 microprocessor is an enhanced version of the 80286 microprocessor that includes a memory management unit, enhanced to provide memory paging. The 80386 also includes 32-bit extended registers and 32-bit address and data buses. A scaled-down version of the 80386DX with a 16-bit data and 24-bit address bus is available as the 80386SX microprocessor. The 80386EX is a complete AT-style personal computer on a chip.
2. The 80386 has a physical memory size of 4G bytes that can be addressed as a virtual memory with up to 64T bytes. The 80386 memory is 32 bits wide and is addressed as bytes, words, or doublewords.
3. When the 80386 is operated in the pipelined mode, it sends the address of the next instruction or memory data to the memory system prior to completing the execution of the current instruction. This allows the memory system to begin fetching the next instruction or data before the current one is completed. This increases access time, thus reducing the speed of the memory.
4. A cache memory system allows data that are frequently read to be accessed in less time because they are stored in high-speed semiconductor memory. If data are written to memory, they are also written to the cache so the most current data are always present in the cache.
5. The I/O structure of the 80386 is almost identical to the 80286, except that I/O can be inhibited when the 80386 is operated in the protected mode, through the I/O bit protection map stored with the TSS.
6. The register set of the 80386 contains extended versions of the registers introduced on the 80286 microprocessor. These extended registers include: EAX, EBX, ECX, EDX, EBP, ESP, EDI, ESI, EIP, and EFLAGS. In addition to the extended registers, two supplemental segment registers (FS and GS) are added. Debug registers

and control registers handle system debugging tasks and memory management in the protected mode.

7. The instruction set of the 80386 is enhanced to include instructions that address the 32-bit extended register set. The enhancements also include additional addressing modes that allow any extended register to address memory data. Scaling has been added so an index register can be multiplied by 1, 2, 4, or 8. New instruction types include bit scan, string moves with sign or zero extension, set byte upon condition, and double-precision shifts.
8. Interrupts in the 80386 microprocessor have been expanded to include additional predefined interrupts in the interrupt vector table. These additional interrupts are used with the memory management system.
9. The 80386 memory manager is similar to the 80286, except the physical addresses generated by the MMU are 32 bits wide instead of 24. The 80386 MMU is also capable of paging.
10. The 80386 is operated in the real mode (8086 mode) when it is reset. The real mode allows the microprocessor to address data in the first 1M byte of memory. In the protected mode, the 80386 addresses any location in its 4G bytes of physical address space.
11. A descriptor is a series of 8 bytes that specifies how a code or data segment is used by the 80386. The descriptor is selected by a selector stored in one of the segment registers. Descriptors are only used in the protected mode.
12. Memory management is accomplished through a series of descriptors stored in descriptor tables. To facilitate memory management, the 80386 uses three descriptor tables: the global descriptor table (GDT), the local descriptor table (LDT), and the interrupt descriptor table (IDT). The GDT and LDT each hold up to 8,192 descriptors, while the IDT holds up to 256 descriptors. The GDT and LDT describe code and data segments and tasks. The IDT describes the 256 different interrupt levels through interrupt gate descriptors.
13. The TSS (task state segment) contains information about the current task and also the previous task. Appended to the end of the TSS is an I/O bit protection map that inhibits selected I/O port addresses.
14. The memory paging mechanism allows any 4K byte physical memory page to be mapped to any 4K byte linear memory page. For example, memory location 00A00000H can be assigned to memory location A0000000H through the paging mechanism. A page directory and page tables are used to assign any physical address to any linear address. The paging mechanism can be used in the protected mode or the virtual mode.
15. The 80486 microprocessor is an improved version of the 80386 microprocessor that contains an 8K byte cache and an 80387 arithmetic coprocessor, and executes many instructions in one clocking period.
16. The 80486 microprocessor executes a few new instructions that control the internal cache memory and allow addition (XADD) and comparison (CMPXCHG) with an exchange and a byte swap (BSWAP) operation. Other than these few additional instructions, the 80486 is 100 percent compatible with the 80386 and 80387.
17. A new feature in the 80486 is the BIST (built-in self-test) that tests the microprocessor, coprocessor, and cache at reset time. If the 80486 passes the test, EAX contains a zero.

18. Additional test registers are added to the 80486 to allow the cache memory to be tested. These new test registers are TR3 (cache data), TR4 (cache status), and TR5 (cache control). These registers are used each time a BIST is performed after a reset operation.

## 9–9 QUESTIONS AND PROBLEMS

1. The 80386 microprocessor addresses _____ bytes of physical memory when operated in the protected mode.
2. The 80386 microprocessor addresses _____ bytes of virtual memory through its memory management unit.
3. Describe the differences between the 80386DX and the 80386SX.
4. Draw the memory map of the 80386 when operated in the
   (a) protected mode
   (b) real mode
5. How much current is available on various 80386 output pin connections? Compare these currents with the currents available at the output pin connection of an 80286 microprocessor.
6. Describe the 80386 memory system and explain the purpose and operation of the bank selection signals.
7. Explain the action of a hardware reset on the address bus connections of the 80386.
8. Explain how pipelining lengthens the access time for many memory references in the 80386 microprocessor-based system.
9. Briefly describe how the cache memory system functions.
10. I/O ports in the 80386 start at I/O address _____ and extend to I/O address _____.
11. What I/O ports communicate data between the 80386 and its companion 80387 coprocesssor?
12. Compare and contrast the memory and I/O connections found in the 80386 with those found in earlier microprocessors.
13. If the 80386 operates at 20 MHz, what clocking frequency is applied to the CLK2 pin?
14. What is the purpose of the $\overline{\text{BS16}}$ pin on the 80386 microprocessor?
15. What two segment registers found in the 80386 programming model are not present in the 80286?
16. List the extended registers found in the 80386 microprocessor.
17. List each 80386 flag register bit and describe its purpose.
18. Define the purpose of each of the control registers (CR0, CR1, CR2, and CR3) in the 80386.
19. Define the purpose of each 80386 debug register.
20. The debug registers cause which level of interrupt?
21. Describe the operation of the bit scan forward instruction.
22. Describe the operation of the bit scan reverse instruction.
23. Describe the operation of the SHRD instruction.
24. Form an instruction that accesses data in the FS segment at the location indirectly addressed by the DI register. The instruction should store the contents of EAX into this memory location.

25. What is scaled-index addressing?
26. Is the following instruction legal? MOV AX,[EBX+ECX]
27. Explain how the following instructions calculate the memory address:
    (a) ADD [EBX+8*ECX],AL
    (b) MOV DATA[EAX+EBX],CX
    (c) SUB EAX,DATA
    (d) MOV ECX,[EBX]
28. What is the purpose of interrupt type 7?
29. Which interrupt vector type is activated for a protection privilege violation?
30. What is a double interrupt fault?
31. If an interrupt occurs in the protected mode, what defines the interrupt vectors?
32. What is a descriptor?
33. What is a selector?
34. How does the selector choose the local descriptor table?
35. What register is used to address the global descriptor table?
36. How many global descriptors can be stored in the GDT?
37. Explain how the 80386 can address a virtual memory space of 64T bytes when the physical memory contains only 4G bytes of memory.
38. What is the difference between a segment descriptor and a system descriptor?
39. What is the task state segment (TSS)?
40. How is the TSS addressed?
41. Describe how the 80386 switches from the real mode to the protected mode.
42. Describe how the 80386 switches from the protected mode to the real mode.
43. What is virtual 8086 mode operation of the 80386 microprocessor?
44. How is the paging directory located by the 80386?
45. How many bytes are found in a page of memory?
46. Explain how linear memory address D0000000H can be assigned to physical memory address C0000000H with the paging unit of the 80386.
47. What are the differences between an 80386 and 80486 microprocessor?
48. What is the purpose of the $\overline{\text{FLUSH}}$ input pin on the 80486 microprocessor?
49. Compare the register set of the 80386 with the 80486 microprocessor.
50. What differences exist between the flags of the 80486 and the flags of the 80386 microprocessor?
51. What pins are used for parity checking on the 80486 microprocessor?
52. The 80486 microprocessor uses _____ parity.
53. The cache inside the 80486 microprocessor is _____ K bytes.
54. A cache line is filled by reading _____ bytes from the memory system.
55. What is an 80486 burst?
56. Define the term cache write-through.
57. What is a BIST?
58. Can 80486 caching be disabled by software? (Explain your answer.)
59. Explain how the XADD EBX,EDX instruction operates.
60. The CMPXCHG CL,AL instruction compares CL with AL. What else occurs when this instruction executes?
61. Compare the INVD instruction with the WBINVD instruction.
62. What is the purpose of the PCD bit in the page table directory or page table entry?
63. Does the PWT bit in the page table directory or page table entry affect the on-chip cache?

# CHAPTER 10

# Memory Interface

## INTRODUCTION

Whether simple or complex, every microprocessor-based system has a memory. The Intel family of microprocessors is no different.

Almost all systems contain two main types of memory, *read-only memory* (ROM) and *random access memory* (RAM) or read/write memory. Read-only memory stores system software and permanent data, while RAM contains temporary data and application software. This chapter explains how to interface both memory types to the Intel family of microprocessors. We demonstrate memory interface to an 8-, 16-, 32-, and 64-bit data bus using various memory address sizes. This allows virtually any microprocessor to be interfaced to any memory system.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Decode the memory address and use the outputs of the decoder to select various memory components.
2. Use programmable logic devices (PLDs) to decode memory addresses.
3. Explain how to interface both RAM and ROM to a microprocessor.
4. Explain how parity can detect memory errors.
5. Interface memory to an 8-, 16-, 32-, and 64-bit data bus.
6. Explain the operation of an interleaved memory system.
6. Explain the operation of a dynamic RAM controller.
7. Interface dynamic RAM to the microprocessor.

# 10–1 MEMORY DEVICES

Before attempting to interface memory to the microprocessor, it is essential to completely understand the operation of memory components. In this section, we explain the function of the three common types of memory: *read-only memory* (ROM), *static random access memory* (SRAM), and *dynamic random access memory* (DRAM).

## Memory Pin Connections

The pin connections common to all memory devices are the address inputs, data outputs or input/outputs, one or more selection inputs, and at least one control input used to select a read or write operation. See Figure 10–1 for ROM and RAM generic memory devices.

*Address Connections.* All memory devices have address inputs that select a memory location within the memory device. The number of address pins found on a memory device is determined by the number of memory locations found within it.

Today, the more common memory devices have between 1K (1,024) and 4M (4,194,304) memory locations. Devices with 16M memory locations are beginning to appear, and 256M memory location devices are on the horizon. A 1K memory device has 10 address pins ($A_0$–$A_9$); therefore, 10 address inputs are required to select any of its 1,024 memory locations. It takes a 10-bit binary number (1,024 different combinations) to select a single location on a 1,024-location device. If a memory device has 11 address connections ($A_0$–$A_{10}$), it has 2,048 (2K) internal memory locations. The number of memory locations can thus be extrapolated from the number of address pins. For example, a 4K memory device has 12 address connections, an 8K device has 13, and so forth. A device that contains 1M locations requires a 20-bit address ($A_0$–$A_{19}$).

400H represents a 1K byte section of the memory system. Thus, if a memory device is decoded to begin at memory address 10000H, and it is a 1K device, its last location is at address 103FFH—one location less than 400H. Another important hexadecimal number

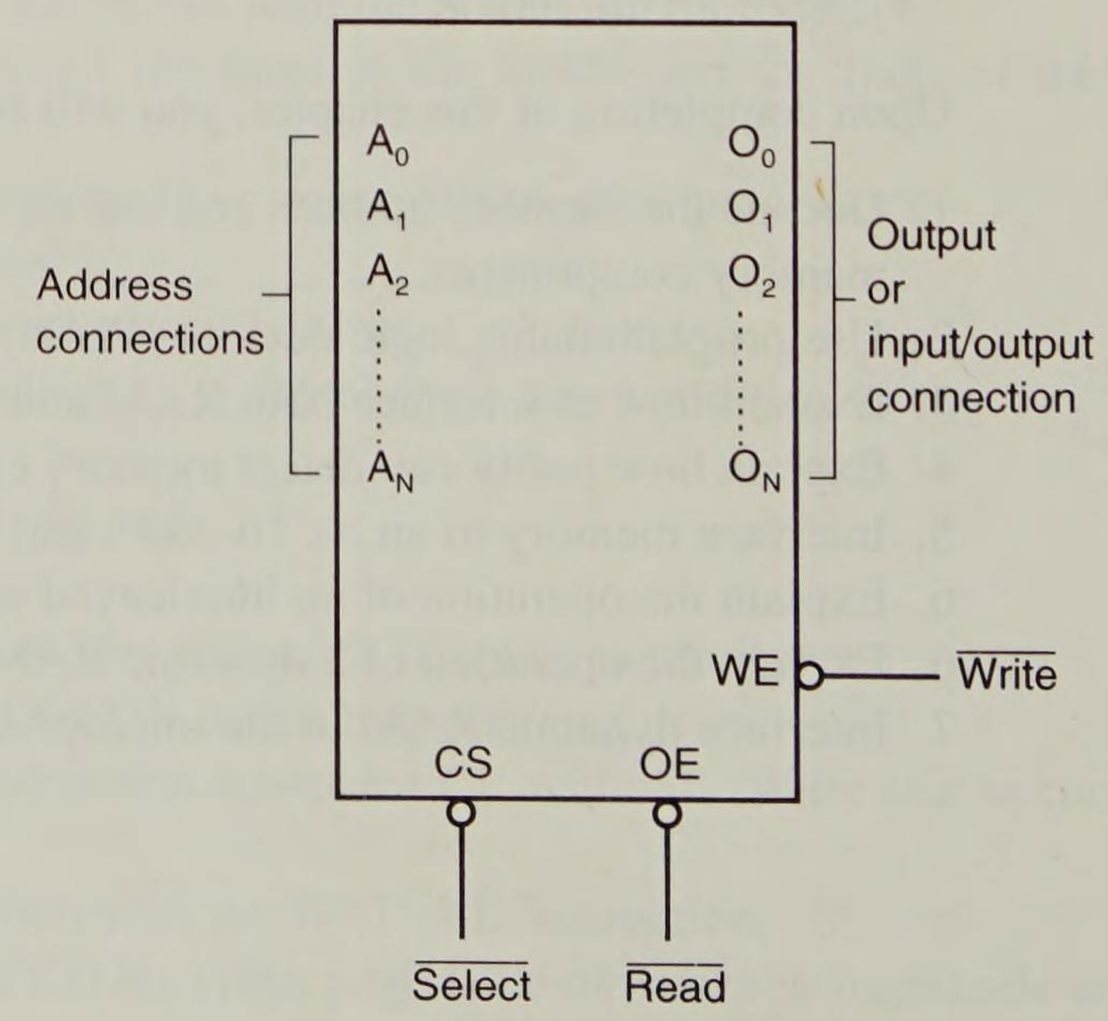



**FIGURE 10–1** A pseudo-memory component illustrating the address, data, and control connections.

to remember is 1000H, because 1000H is 4K. A memory device that contains a starting address of 14000H and is 4K bytes in size ends at location 14FFFH—one location less than 1000H. A third number is 64K or 10000H. A memory that starts at location 30000H and ends at location 3FFFFH is a 64K byte memory. Finally, a 1M byte memory contains 100000H locations. A 1M byte memory that starts at location 02000000H in a Pentium or similar microprocessor ends at location 020FFFFFH.

*Data Connections.* All memory devices have a set of data outputs or input/outputs. The device illustrated in Figure 10–1 has a common set of input/output (I/O) connections. Today many memory devices have bidirectional common I/O pins used for both reading and writing data.

The data connections are the points at which data are entered for storage or extracted for reading. In the sample memory device shown in Figure 10–1, there are eight I/0 connections, which means that the memory device stores 8 bits of data in each of its memory locations. An 8-bit-wide memory device is often called a *byte-wide* memory. Although most memory devices are currently 8 bits wide, not all are. Some devices are 16 bits, 4 bits, or just 1 bit wide.

Catalog listings of memory devices often refer to memory size as the number of memory locations times the number of bits per location. For example, a memory device with 1K memory locations with 8 bits in each location is often listed as a 1K $\times$ 8 by the manufacturer. A 16K $\times$ 1 memory device contains 16K memory locations with 1 bit in each location. Memory devices are also classified according to total bit capacity. For example, a 1K $\times$ 8-bit memory device is sometimes listed as an 8K memory device, or a 64K $\times$ 4-bit memory device is listed as a 256K device. These variations occur from one manufacturer's data sheet to another's.

*Selection Connections.* Each memory device has an input—sometimes more than one—that selects or enables the memory device. This kind of input is most often called a *chip select* ($\overline{\text{CS}}$), *chip enable* ($\overline{\text{CE}}$), or *select* ($\overline{\text{S}}$) input. RAM memory generally has at least one $\overline{\text{CS}}$ or $\overline{\text{S}}$ input, and ROM at least one $\overline{\text{CE}}$. If the $\overline{\text{CE}}$, $\overline{\text{CS}}$, or $\overline{\text{S}}$ input is active (a logic 0 is active in this case, because of the overbar), the memory device performs a read or a write; if it is inactive (a logic 1 is inactive in this case because of the overbar), the memory device cannot do a read or a write because it is turned off or disabled. (Note that some manufacturers have recently started using the # symbol to indicate an active low input. For example, the CS# symbol is equivalent to $\overline{\text{CS}}$.) If more than one $\overline{\text{CS}}$ connection is present, all must be activated to read or write data. Memory access time is determined from the point where the select input is activated until the data appear at the data output connection. This assumes that an address is applied to the address connections at the same time as the select input is activated. In some devices, access time is measured from the last change on the address and selection input.

*Control Connections.* All memory devices have some form of control input. A ROM usually has only one control input, while a RAM often has one or two control inputs.

The control input most often found on a ROM is the *output enable* ($\overline{\text{OE}}$) or *gate* ($\overline{\text{G}}$) connection, which allows data to flow out of the output data pins of the ROM. If $\overline{\text{OE}}$ and the selection input are both active, then the outputs are enabled; if $\overline{\text{OE}}$ is inactive, the outputs are disabled at their high-impedance state. The $\overline{\text{OE}}$ connection enables and disables a set of three-state buffers located within the memory device and must be active to read data.

A RAM memory device has either one or two control inputs. If there is one control input, it is often called R/$\overline{W}$. This pin selects a read operation or a write operation only if the device is selected by the selection input ($\overline{CS}$). If the RAM has two control inputs, they are usually labeled $\overline{WE}$ (or $\overline{W}$) and $\overline{OE}$ (or $\overline{G}$). The $\overline{WE}$ (*write enable*) pin must be active to perform a memory write, and $\overline{OE}$ must be active to perform a memory read operation. Whenever these two controls ($\overline{WE}$ and $\overline{OE}$) are present, they must never both be active at the same time. If both control inputs are inactive (logic 1s), then data are neither written nor read and the data connections are at their high-impedance state.

## ROM Memory

The read-only memory (ROM) permanently stores programs and data that are resident to the system and must not change when power is disconnected. The ROM is permanently programmed so data is always present, even when power is disconnected.

The ROM is available in many forms today. A device we call a ROM is purchased in mass quantities from a manufacturer and programmed during its fabrication at the factory. The EPROM (*erasable programmable read-only memory*), a type of ROM, is more commonly used when software must be changed often or when the number of ROMS needed is too small to be economical. For a ROM to be practical, we usually must purchase at least 10,000 devices. An EPROM is programmed in the field on a device called an EPROM programmer. The EPROM is also erasable if exposed to high-intensity ultraviolet light for about 10 to 20 minutes, depending on the type of EPROM.

PROM memory devices are also available, but they are not as common today. The PROM (*programmable read-only memory*) is also programmed in the field. Programming is accomplished by burning open tiny Nichrome or silicon oxide fuses, but once programmed, the PROM cannot be erased.

Another, newer, type of *read-mostly* memory (RMM) is called the flash memory.* The flash memory is also often called an EEPROM (electrically erasable programmable ROM), EAROM (electrically alterable ROM), or a NOVRAM (nonvolatile ROM). The flash memory device is electrically erasable in the field, but requires more time to erase than a normal RAM. The typical erase time is 10 ms per byte. The flash memory device is used to store setup information for systems such as the video adapter card in the computer. It is also replacing the EPROM in the computer for the BIOS memory. Some systems contain a password stored in a flash memory device. Be aware that flash memory has a limited life of 100,000 erase cycles. For this reason, and for its slow write speed, flash memory will not replace other read/write memory in the near future.

Flash memory operates from a single 5.0 V power supply and most devices are pin compatible with static RAMs. For example, the 8K × 8, μPD28C64† is pin compatible with the 8K × 8, 6164 static RAM. The μPD28C64 requires 10 ms whenever a byte is erased and rewritten. The write operation is identical to that of an SRAM, except for the cycle time of 10 ms. Care must be taken so another write cycle does not occur within this

---

*Flash memory is a registered trademark of Intel Corporation.

†A product of NEC Electronics, Inc.

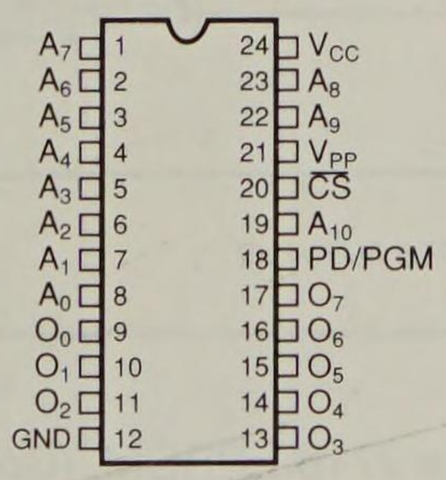
10 ms interval. This is usually handled with a software time delay. If a 15 V power supply is available, the entire chip can be erased at one time by placing a 15 V pulse on the $\overline{OE}$ connection, logic ones on the data connections, and a logic zero on both $\overline{CE}$ and $\overline{WE}$ for 10 ms.

Figure 10–2 illustrates the 2716 EPROM, a member of a much larger family of EPROMs. This device contains 11 address inputs and 8 data outputs. The 2716 is a 2K × 8 memory device. The 27XXX series of EPROMs contains the following part numbers: 2704 (512 × 8), 2708 (1K × 8), 2716 (2K × 8), 2732 (4K × 8), 2764 (8K × 8), 27128 (16K × 8), 27256 (32K × 8), 27512 (64K × 8), and 271024 (128K × 8). Each of these parts contains address pins, 8 data connections, one chip selection input (CE), and an output enable pin (OE). Notice that if the size is multiplied by the number of bits in a word, the part number is found. For example, the 8K × 8 EPROM is a 64K memory device, the *2764*.

Figure 10–3 illustrates the timing diagram for the 2716 EPROM. Data only appear on the output connections after a logic 0 is placed on both the $\overline{CE}$ and $\overline{OE}$ pin connections. If $\overline{CE}$ and $\overline{OE}$ are not both logic 0s, the data output connections remain at their high-impedance, or off, states. Note that the Vpp pin must be placed at a logic 1 for data to be read from the EPROM. In some cases, the Vpp pin is in the same position as the $\overline{WE}$ pin on the SRAM. This can allow a single socket to hold either an EPROM or an SRAM. An example is the 2716 EPROM and the 6116 SRAM, both 2K × 8 devices that have the same pinout except for Vpp on the EPROM and $\overline{WE}$ on the SRAM.

One important piece of information provided by the timing diagram and data sheet is the memory access time—the time that it takes the memory to read information. As Figure 10–3 illustrates, memory access time ($T_{acc}$) is measured from the appearance of the address at the address inputs until the appearance of the data at the output connections.

PIN CONFIGURATION




PIN NAMES

| | |
|---|---|
| $A_0$–$A_{10}$ | ADDRESSES |
| PD/PGM | POWER DOWN/PROGRAM |
| $\overline{CS}$ | CHIP SELECT |
| $O_0$–$0_7$ | OUTPUTS |

MODE SELECTION

| PINS / MODE | PD/PGM (18) | $\overline{CS}$ (20) | $V_{PP}$ (21) | $V_{CC}$ (24) | OUTPUTS (9-11, 13-17) |
|---|---|---|---|---|---|
| Read | $V_{IL}$ | $V_{IL}$ | +5 | +5 | $D_{OUT}$ |
| Deselect | Don't Care | $V_{IH}$ | +5 | +5 | High Z |
| Power down | $V_{IH}$ | Don't Care | +5 | +5 | High Z |
| Program | Pulsed $V_{IL}$ to $V_{IH}$ | $V_{IH}$ | +25 | +5 | $D_{IN}$ |
| Program Verify | $V_{IL}$ | $V_{IL}$ | +25 | +5 | $D_{OUT}$ |
| Program Inhibit | $V_{IL}$ | $V_{IH}$ | +25 | +5 | High Z |

BLOCK DIAGRAM

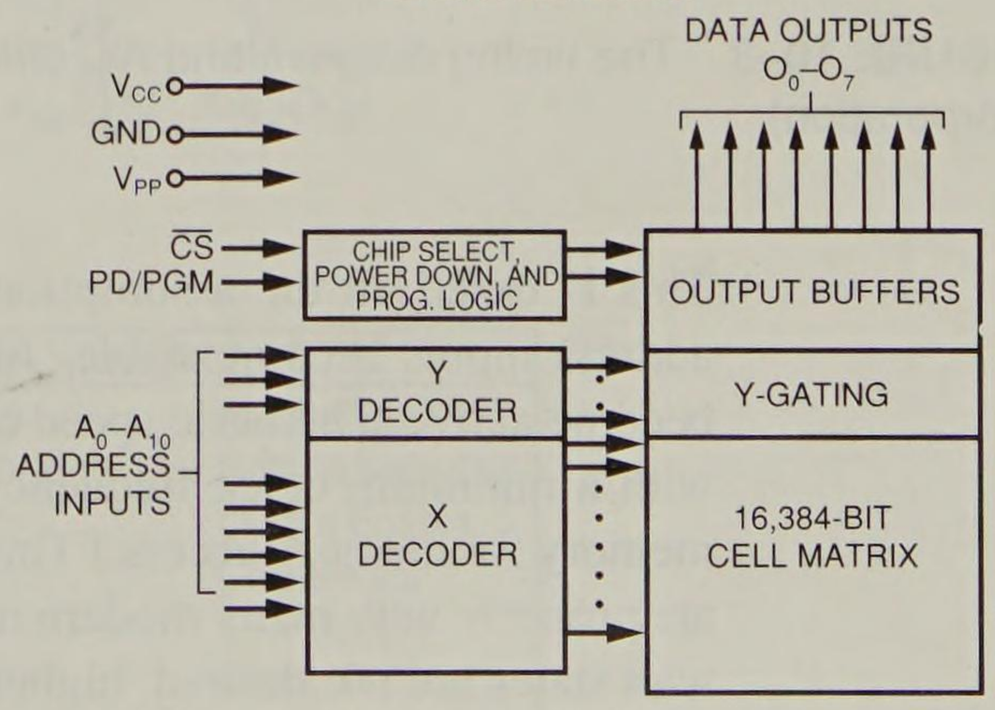



**FIGURE 10–2** The pinout of the 2716, 2K × 8 EPROM. (Courtesy of Intel Corporation)

A.C. Characteristics

$T_A$ = 0°C to 70°C, $V_{CC}$[1] = +5V ±5%, $V_{PP}$[2] = $V_{CC}$ ±0.6V[3]

| Symbol | Parameter | Limits | | | Unit | Test Conditions |
|---|---|---|---|---|---|---|
| | | Min. | Typ.[4] | Max. | | |
| $t_{ACC1}$ | Address to Output Delay | | 250 | 450 | ns | PD/PGM = $\overline{CS}$ = $V_{IL}$ |
| $t_{ACC2}$ | PD/PGM to Output Delay | | 280 | 450 | ns | $\overline{CS}$ = $V_{IL}$ |
| $t_{CO}$ | Chip Select to Output Delay | | | 120 | ns | PD/PGM = $V_{IL}$ |
| $t_{PF}$ | PD/PGM to Output Float | 0 | | 100 | ns | $\overline{CS}$ = $V_{IL}$ |
| $t_{DF}$ | Chip Deselect to Output Float | 0 | | 100 | ns | PD/PGM = $V_{IL}$ |
| $t_{OH}$ | Address to Output Hold | 0 | | | ns | PD/PGM = $\overline{CS}$ = $V_{IL}$ |

Capacitance[1] $T_A$ = 25°C, f = 1 MHz

| Symbol | Parameter | Typ. | Max. | Unit | Conditions |
|---|---|---|---|---|---|
| $C_{IN}$ | Input Capacitance | 4 | 6 | pF | $V_{IN}$ = 0V |
| $C_{OUT}$ | Output Capacitance | 8 | 12 | pF | $V_{OUT}$ = 0V |

A.C. Test Conditions:
Output Load: 1 TTL gate and $C_L$ = 100 pF
Input Rise and Fall Times: 20 ns
Input Pulse Levels: 0.8V to 2.2V
Timing Measurement Reference Level:
Inputs 1V and 2V
Outputs 0.8V and 2V

WAVEFORMS

A. Read Mode
PD/PGM = $V_{IL}$

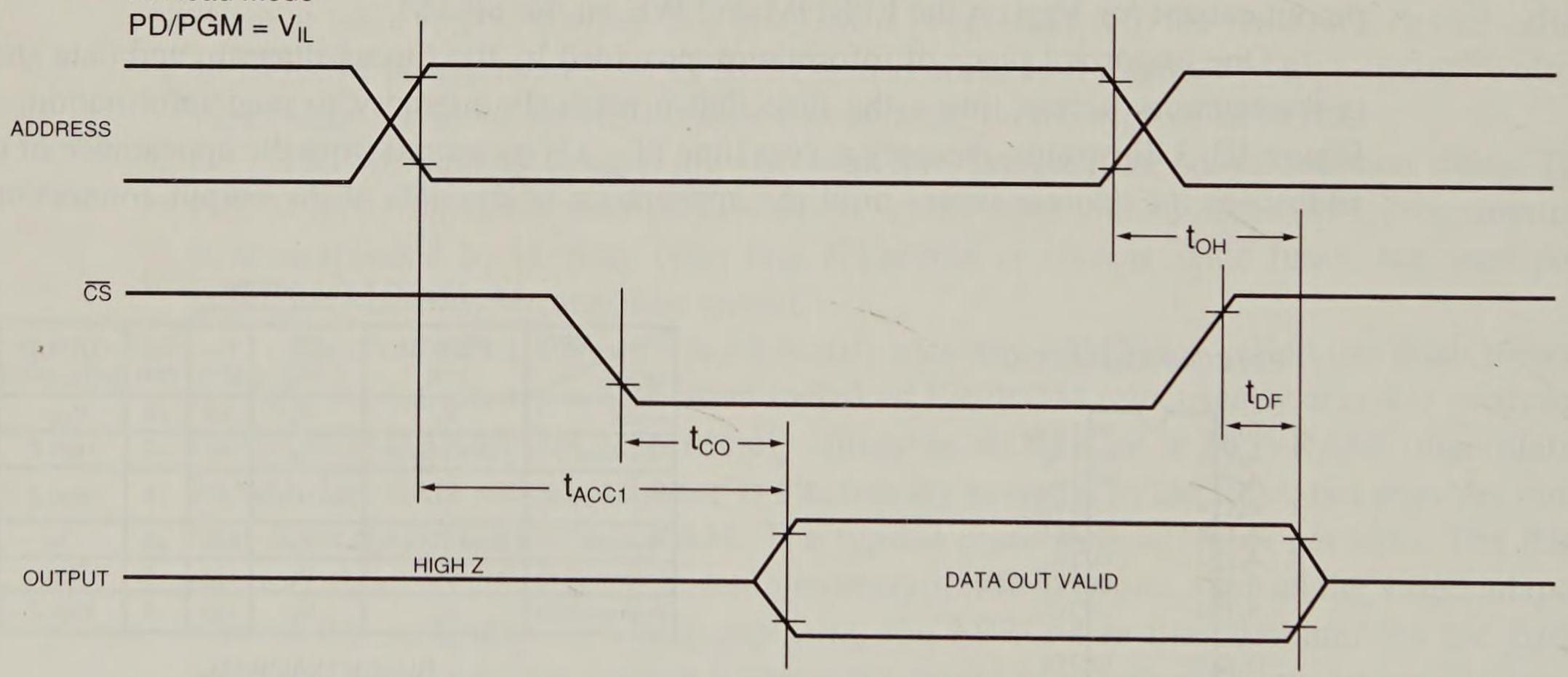



**FIGURE 10–3** The timing diagram and AC characteristics of the 2716 EPROM. (Courtesy of Intel Corporation)

This is based on the assumption that the $\overline{CE}$ input goes low at the same time that the address inputs become stable. Also, $\overline{OE}$ must be a logic 0 for the output connections to become active. The basic speed of this EPROM is 450 ns. (Recall that the 80386 operates with a minimum clock frequency of 16 MHz, and allows much less than 450 ns to the memory for memory access.) This type of memory component requires wait states to operate properly with many modern microprocessors, because of its rather long access time. If wait states are not desired, higher speed versions of the EPROM are available at an additional cost. Today EPROM memory is available with access times of as little as 100 ns.

## Static RAM (SRAM) Devices

Static RAM memory devices retain data for as long as DC power is applied. Because no special action (except power) is required to retain stored data, this device is called a *static memory*. The main difference between a ROM and a RAM is that a RAM is written under normal operation and a ROM is programmed outside the computer and is only normally read. The SRAM stores temporary data and is used when the size of the read/write memory is relatively small. Today, a small memory is one that is less than 1M bytes. The SRAM also finds wide application in computer systems for the external or level 2 cache memory.

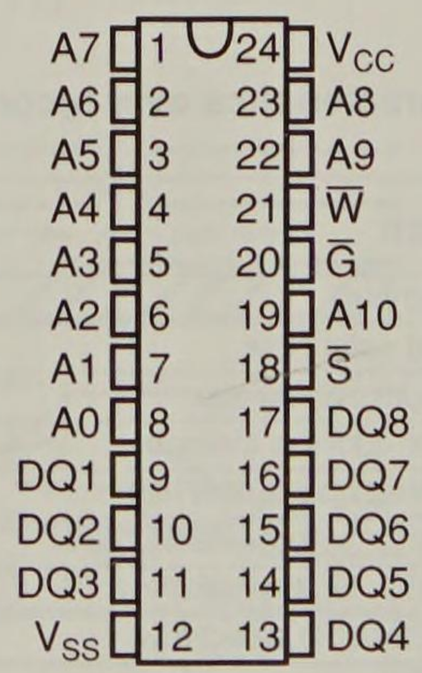
Figure 10–4 illustrates the 4016 SRAM, which is a 2K × 8 read/write memory. This device has 11 address inputs and 8 data input/output connections.

The control inputs of this RAM are slightly different from those presented earlier. The $\overline{OE}$ pin is labeled $\overline{G}$, the $\overline{CS}$ pin is labeled $\overline{S}$, and the $\overline{WE}$ pin is labeled $\overline{W}$. Despite the altered designations, the control pins function exactly the same way as those outlined previously. Other manufacturers make this popular SRAM under the part numbers 2016 and 6116.

Figure 10–5 depicts the timing diagram for the 4016 SRAM. As the read cycle timing reveals, the access time is $t_a$ (A). On the slowest version of the 4016, this time is 250 ns, which requires wait states when connected to an 80386 operated at 16 MHz. Again, it is important to remember that the access time must be checked to determine the compatibility of memory components with the microprocessor. If no wait state operation is desired, SRAM memory is available with access times as short as 15 ns.

**FIGURE 10–4** The pinout of the TMS4016. 2K × 8 static RAM (SRAM). (Courtesy of Texas Instruments Incorporated)

TMS4016 . . . NL PACKAGE
(TOP VIEW)




| PIN NOMENCLATURE | |
|---|---|
| A0 – A10 | Addresses |
| DQ1 – DQ8 | Data In/Data Out |
| $\overline{G}$ | Output Enable |
| $\overline{S}$ | Chip Select |
| $V_{CC}$ | +5 V Supply |
| $V_{SS}$ | Ground |
| $\overline{W}$ | Write Enable |

**Electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)**

| PARAMETER | | TEST CONDITIONS | | MIN | TYP† | MAX | UNIT |
|---|---|---|---|---|---|---|---|
| $V_{OH}$ | High level voltage | $I_{OH}$ = −1 mA, | $V_{CC}$ = 4.5 V | 2.4 | | | V |
| $V_{OL}$ | Low level voltage | $I_{OL}$ = 2.1 mA, | $V_{CC}$ = 4.5 V | | | 0.4 | V |
| $I_I$ | Input current | $V_I$ = 0 V to 5.5 V | | | | 10 | μA |
| $I_{OZ}$ | Off-state output current | $\overline{S}$ or $\overline{G}$ at 2 V or $\overline{W}$ at 0.8 V, $V_O$ = 0 V to 5.5 V | | | | 10 | μA |
| $I_{CC}$ | Supply current from $V_{CC}$ | $I_O$ = 0 mA, $T_A$ = 0°C (worst case) | $V_{CC}$ = 5.5 V, | | 40 | 70 | mA |
| $C_i$ | Input capacitance | $V_I$ = 0 V, | f = 1 MHz | | | 8 | pF |
| $C_O$ | Output capacitance | $V_O$ = 0 V, | f = 1 MHz | | | 12 | pF |

†All typical values are at $V_{CC}$ = 5 V, $T_A$ = 25°C.

**Timing requirements over recommended supply voltage range and operating free-air temperature range**

| PARAMETER | | TMS4016-12 | | TMS4016-15 | | TMS4016-20 | | TMS4016-25 | | UNIT |
|---|---|---|---|---|---|---|---|---|---|---|
| | | MIN | MAX | MIN | MAX | MIN | MAX | MIN | MAX | |
| $t_{c(rd)}$ | Read cycle time | 120 | | 150 | | 200 | | 250 | | ns |
| $t_{c(wr)}$ | Write cycle time | 120 | | 150 | | 200 | | 250 | | ns |
| $t_{w(W)}$ | Write pulse width | 60 | | 80 | | 100 | | 120 | | ns |
| $t_{su(A)}$ | Address setup time | 20 | | 20 | | 20 | | 20 | | ns |
| $t_{su(S)}$ | Chip select setup time | 60 | | 80 | | 100 | | 120 | | ns |
| $t_{su(D)}$ | Data setup time | 50 | | 60 | | 80 | | 100 | | ns |
| $t_{h(A)}$ | Address hold time | 0 | | 0 | | 0 | | 0 | | ns |
| $t_{h(D)}$ | Data hold time | 5 | | 10 | | 10 | | 10 | | ns |

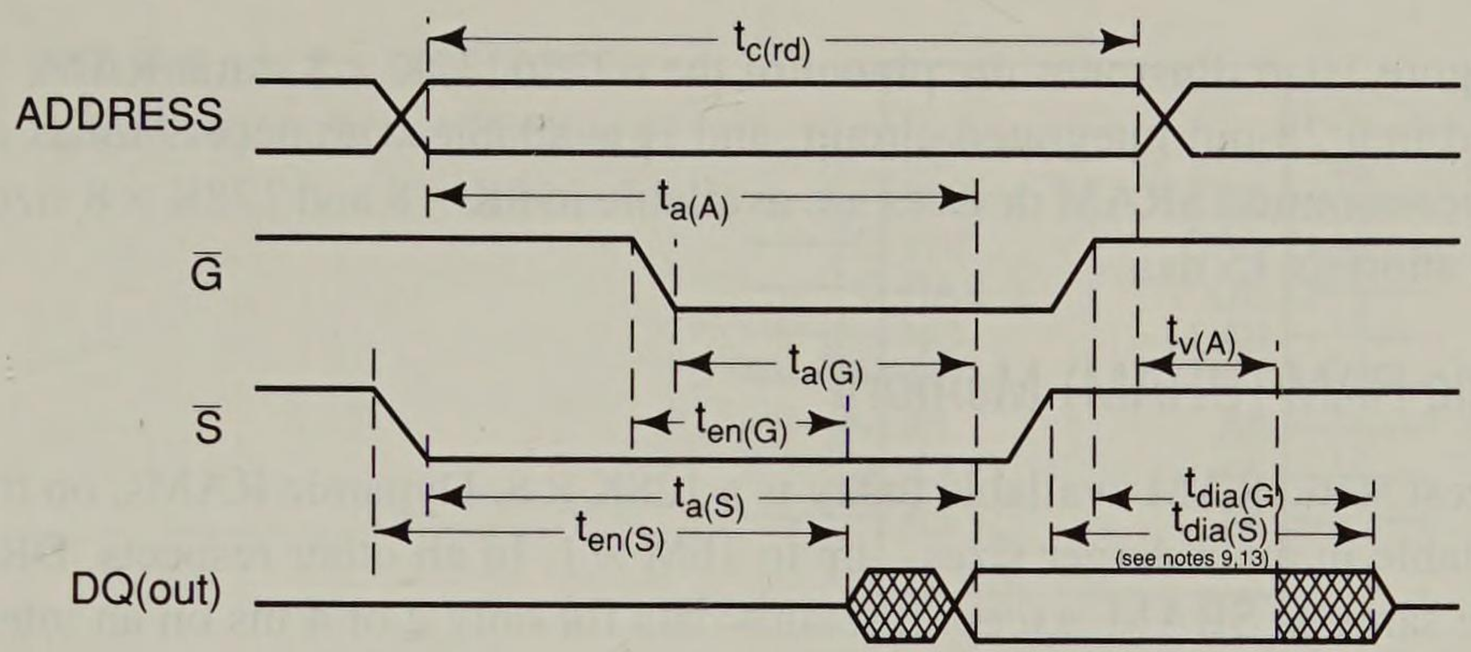
**Switching characteristics over recommended voltage range, $T_A$ = 0°C to 70°C**

| PARAMETER | | TMS4016-12 | | TMS4016-15 | | TMS4016-20 | | TMS4016-25 | | UNIT |
|---|---|---|---|---|---|---|---|---|---|---|
| | | MIN | MAX | MIN | MAX | MIN | MAX | MIN | MAX | |
| $t_{a(A)}$ | Access time from address | | 120 | | 150 | | 200 | | 250 | ns |
| $t_{a(S)}$ | Access time from chip select low | | 60 | | 75 | | 100 | | 120 | ns |
| $t_{a(G)}$ | Access time from output enable low | | 50 | | 60 | | 80 | | 100 | ns |
| $t_{v(A)}$ | Output data valid after address change | 10 | | 15 | | 15 | | 15 | | ns |
| $t_{dis(S)}$ | Output disable time after chip select high | | 40 | | 50 | | 60 | | 80 | ns |
| $t_{dis(G)}$ | Output disable time after output enable high | | 40 | | 50 | | 60 | | 80 | ns |
| $t_{dis(W)}$ | Output disable time after write enable low | | 50 | | 60 | | 60 | | 80 | ns |
| $t_{en(S)}$ | Output enable time after chip select low | 5 | | 5 | | 10 | | 10 | | ns |
| $t_{en(G)}$ | Output enable time after output enable low | 5 | | 5 | | 10 | | 10 | | ns |
| $t_{en(W)}$ | Output enable time after write enable high | 5 | | 5 | | 10 | | 10 | | ns |

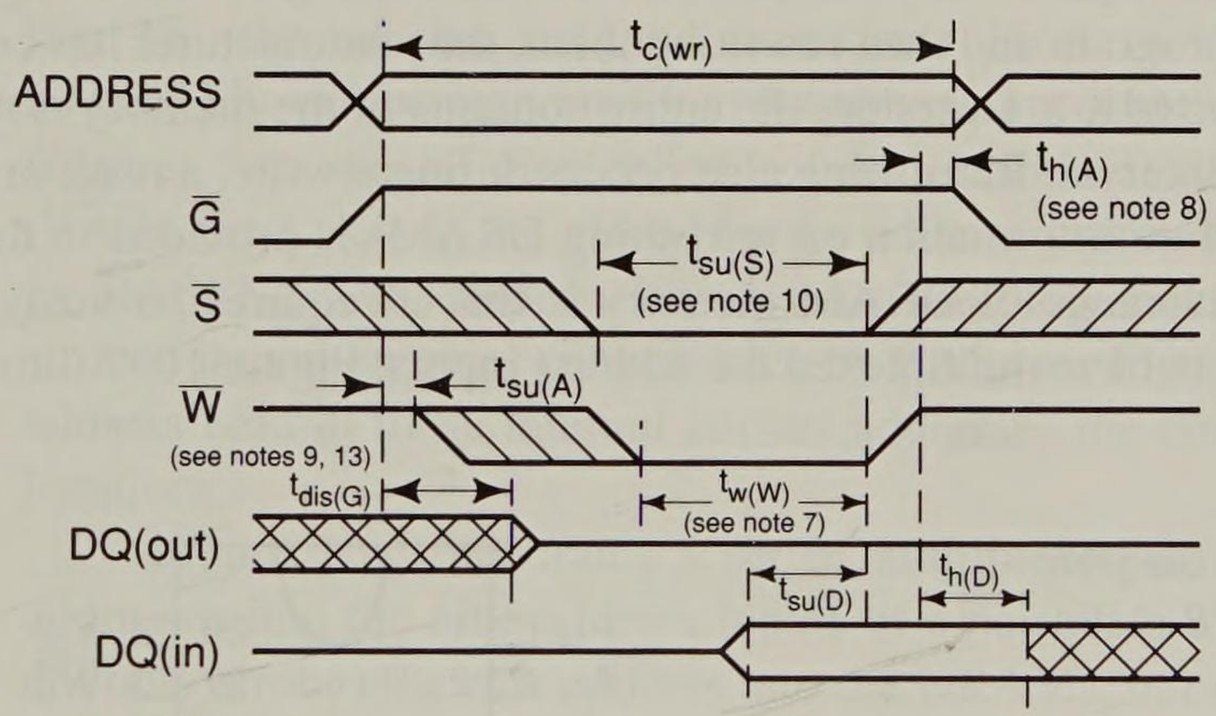
Notes: 3. $C_L$ = 100pF for all measurements except $t_{dis(W)}$ and $t_{en\ (W)}$.
$C_L$ = 5 pF for $t_{dis(W)}$ and $t_{en(W)}$.
4. $t_{dis}$ and $t_{en}$ parameters are sampled and not 100% tested.

**FIGURE 10–5** The AC characteristics of the TMS4016 SRAM (above). The timing diagrams of the TMS4016 SRAM (opposite page). (Courtesy of Texas Instruments Incorporated)

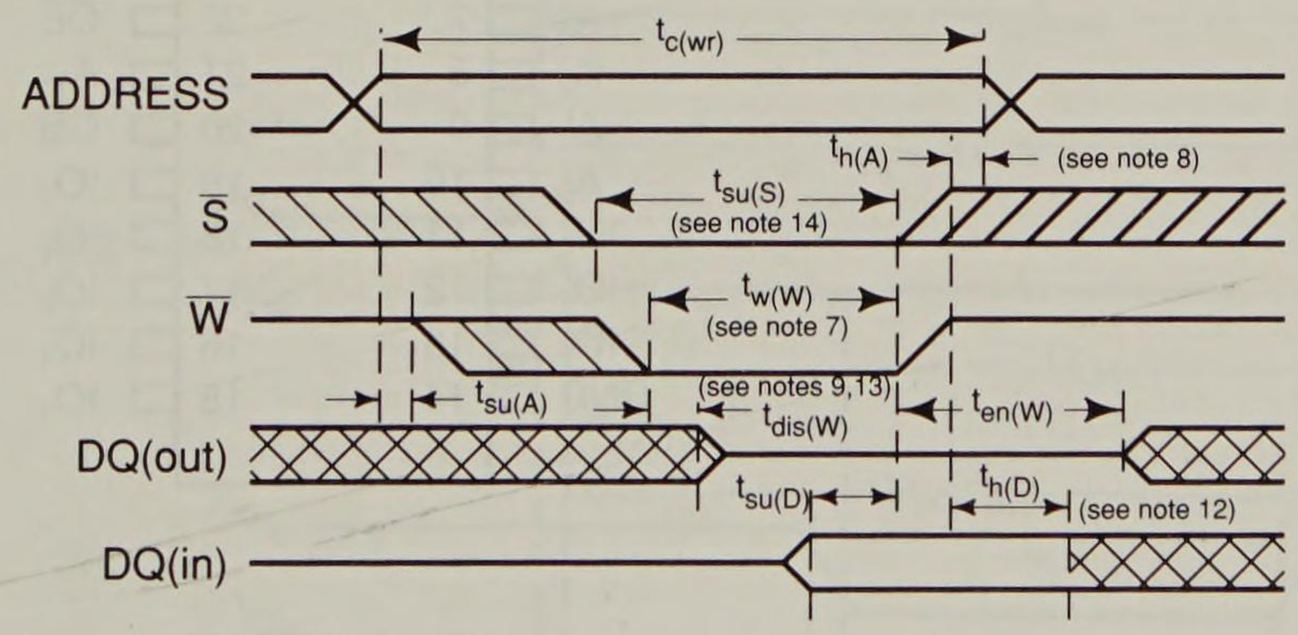
**Timing waveform of read cycle (see note 5)**




**Timing waveform of write cycle no. 1 (see note 6)**




**Timing waveform of write cycle no. 2 (see notes 6 and 11)**




NOTES:
5. $\overline{W}$ is high read Cycle.
6. $\overline{W}$ must be high during all address transitions.
7. A write occurs during the overlap of a low $\overline{S}$ and a low $\overline{W}$.
8. $t_{h(A)}$ is measured from the earlier of $\overline{S}$ or $\overline{W}$ going high to the end of the write cycle.
9. During this period, I/O pins are in the output state so that the input signals of opposite phase to the outputs must not be applied.
10. If the Slow transition occurs simultaneously with the $\overline{W}$ low transitions or after the $\overline{W}$ transition, output remains in a high impedance state.
11. G is continuously low ($G = V_{IL}$).
12. If $\overline{S}$ is low during this period, I/O pins are in the output state. Data input signals of opposite phase to the outputs must not be applied.
13. Transition is measured ± 200 mV from steady-state voltage.
14. If the $\overline{S}$ low transition occurs before the W low transition, then the data input signals of opposite phase to the outputs must not be applied for the duration of $t_{dis(W)}$ after the $\overline{W}$ low transition.

**FIGURE 10–5** (continued)

Figure 10–6 illustrates the pinout of the 62256, 32K × 8 static RAM. This device is packaged in a 28-pin integrated circuit, and is available with access times of 150 ns and less. Other common SRAM devices are available in 8K × 8 and 128K × 8 sizes with access times as short as 15 ns.

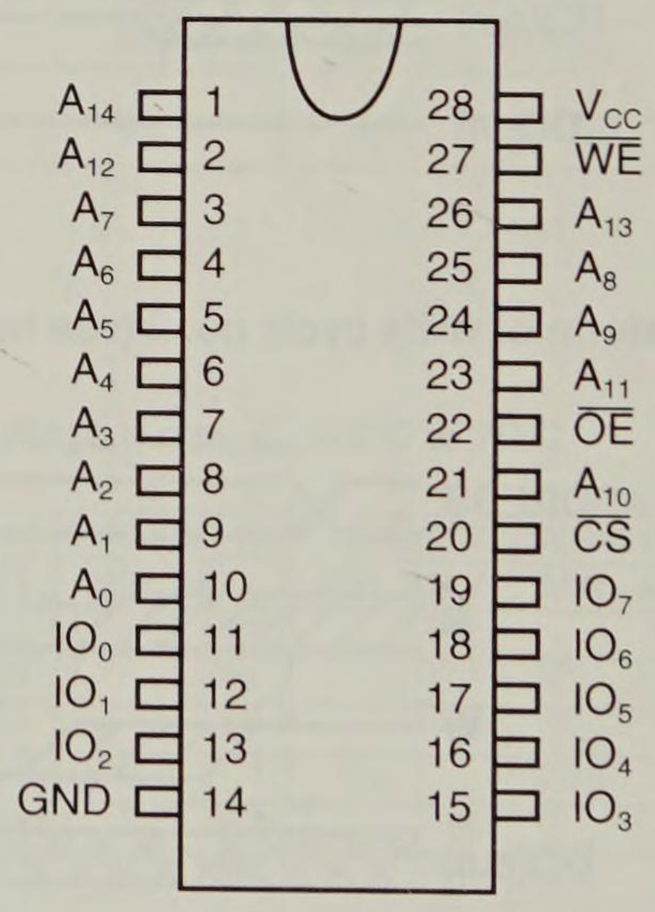
## Dynamic RAM (DRAM) Memory

The largest static RAM available today is a 128K × 8. Dynamic RAMs, on the other hand, are available in much larger sizes—up to 16M × 1. In all other respects, DRAM is essentially the same as SRAM, except it retains data for only 2 or 4 ms on an integrated capacitor. After 2 or 4 ms, the contents of the DRAM must be completely rewritten (*refreshed*) because the capacitors, which store a logic 1 or logic 0, lose their charges.

Instead of requiring the almost impossible task of reading the contents of each memory location with a program and then rewriting them, the manufacturer has constructed the DRAM so that, in the 64K × 1 version, the entire contents of the memory is refreshed with 256 reads in a 4 ms interval. Refreshing also occurs during a write, a read, or during a special refresh cycle. More information on refreshing DRAMs is provided in Section 10–6.

Another disadvantage of DRAM memory is that it requires so many address pins that the manufacturers have multiplexed the address inputs. Figure 10–7 illustrates a 256K

**FIGURE 10–6** Pin diagram of the 62256, 32K × 8 static RAM.




PIN FUNCTION

| | |
|---|---|
| $A_0$–$A_{14}$ | Addresses |
| $IO_0$–$IO_7$ | Data connections |
| $\overline{CS}$ | Chip select |
| $\overline{OE}$ | Output enable |
| $\overline{WE}$ | Write enable |
| $V_{CC}$ | +5V Supply |
| GND | Ground |

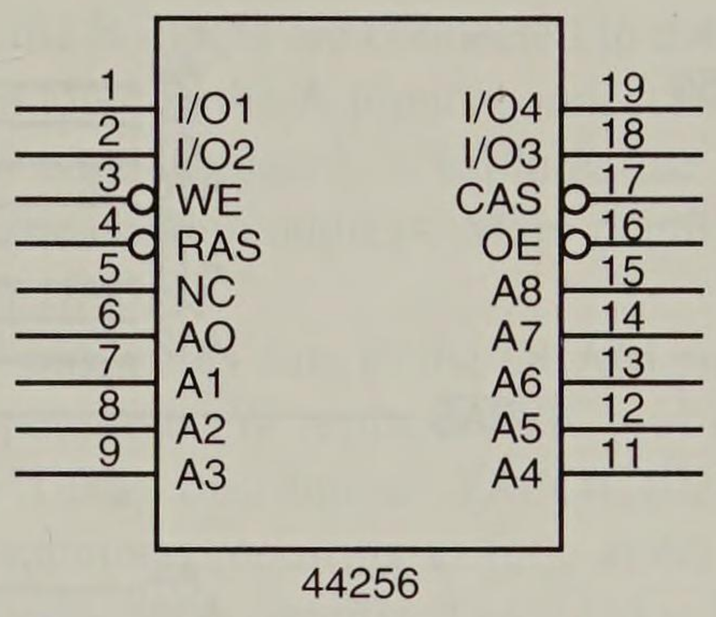



**FIGURE 10–7** The 44256 DRAM organized as 256K × 4.

× 4 DRAM, the 44256. Notice that it contains only 9 address inputs where it should contain 18—the number required to address 256K memory locations. The only way 18 address bits can be crammed into 9 address pins is in two 9-bit increments. This operation requires two special pins called *column address strobe* ($\overline{CAS}$) and *row address strobe* ($\overline{RAS}$). First, $A_0$–$A_8$ are placed on the address pins and strobed into an internal row latch by $\overline{RAS}$. Next, the address bits $A_9$–$A_{17}$ are placed on the same nine address inputs and strobed into an internal column latch by $\overline{CAS}$ (see Figure 10–8 for this timing). The 18-bit address held in these internal latches addresses the contents of one of the 4-bit memory locations.

Figure 10–9 illustrates a set of multiplexers used to strobe the column and row addresses into the nine address inputs of a pair of 44256 DRAMs. Here, the $\overline{RAS}$ signal not only strobes the row address into the DRAMs, but it also changes the address applied to the address inputs. This is possible due to the long propagation delay time of the multi-

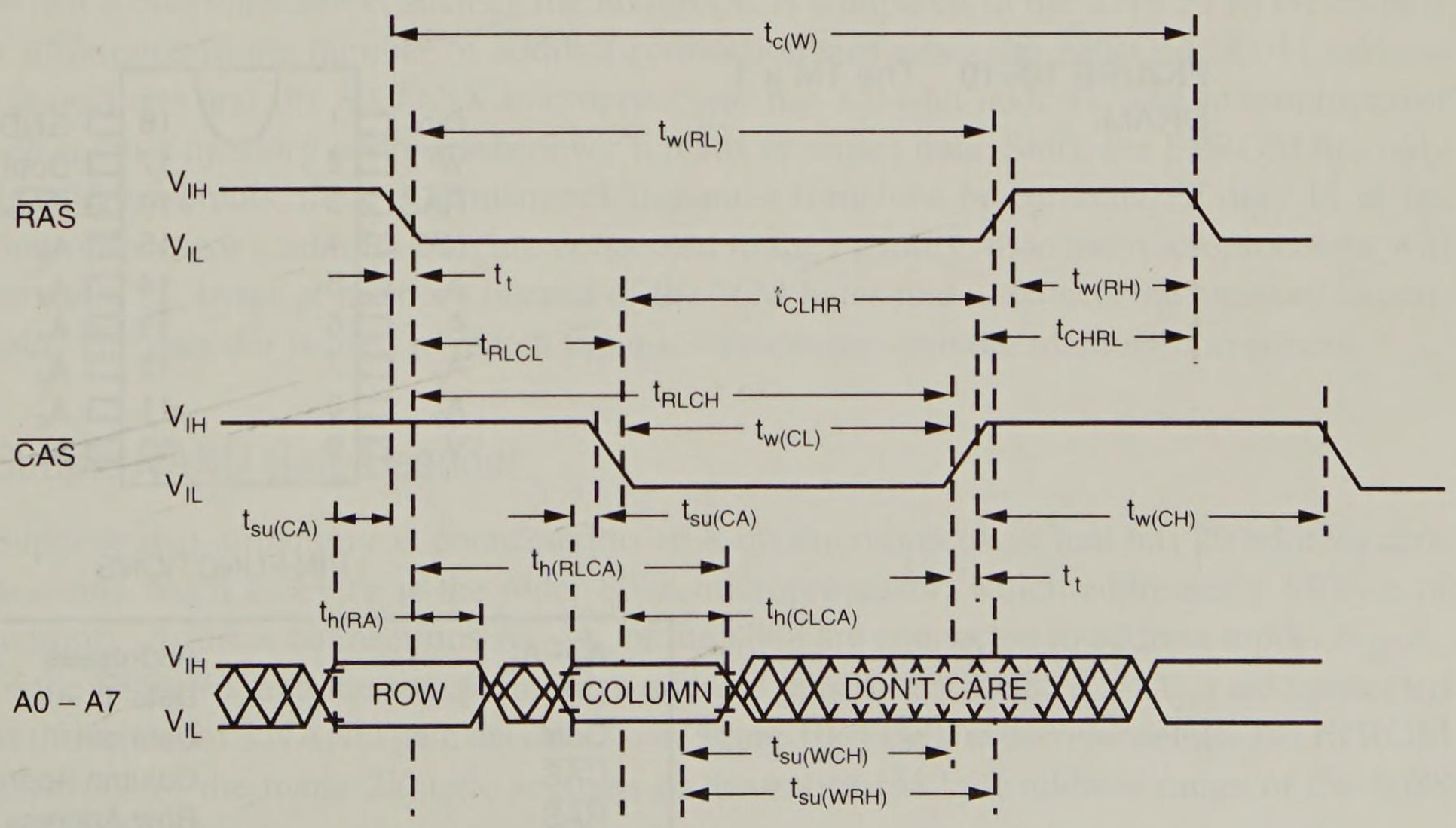



**FIGURE 10–8** RAS, CAS, and address input timing for the TMS44256 DRAM. (Courtesy of Texas Instruments Incorporated)

**FIGURE 10–9** Address multiplexers for the 44256 DRAM.

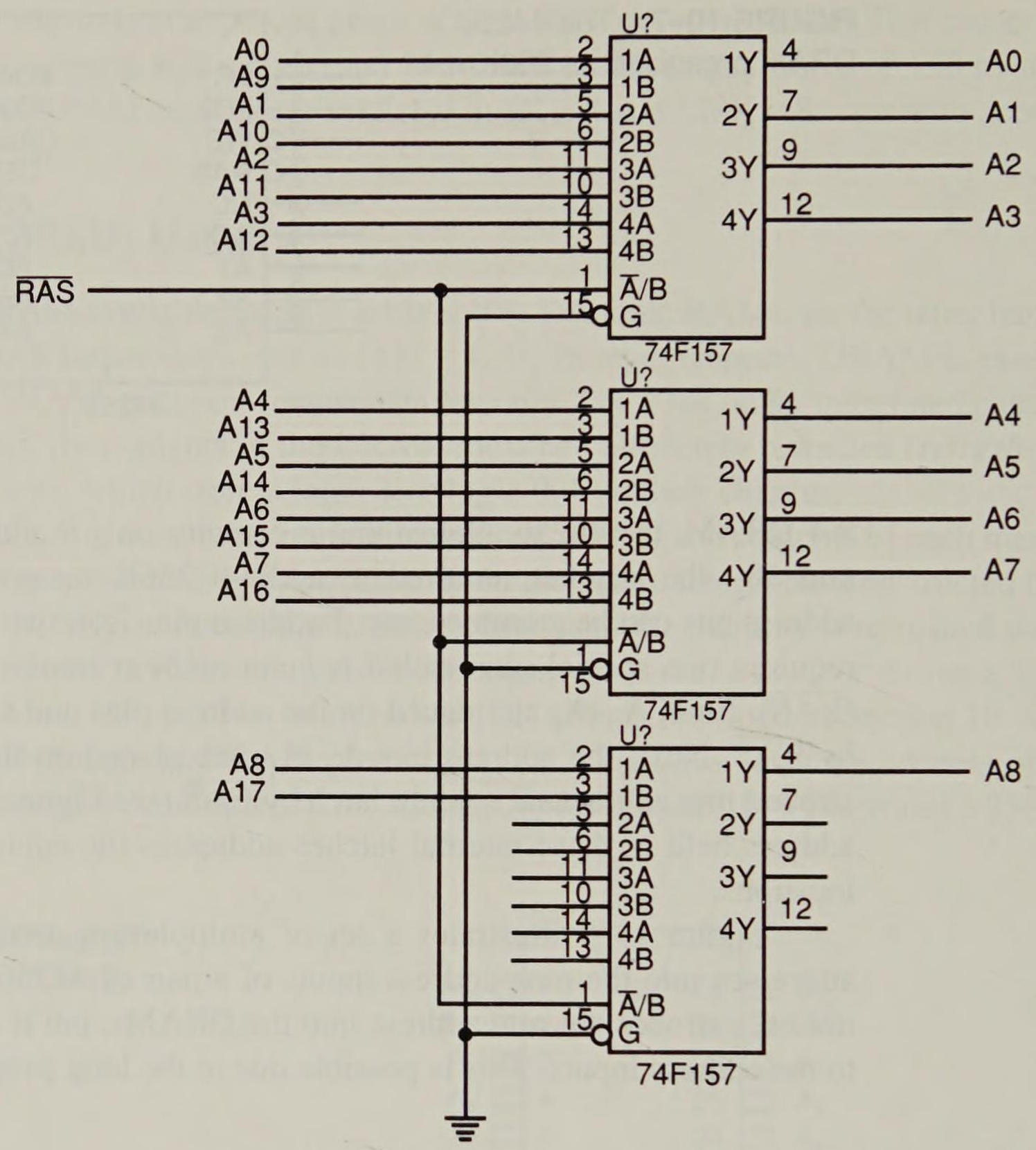
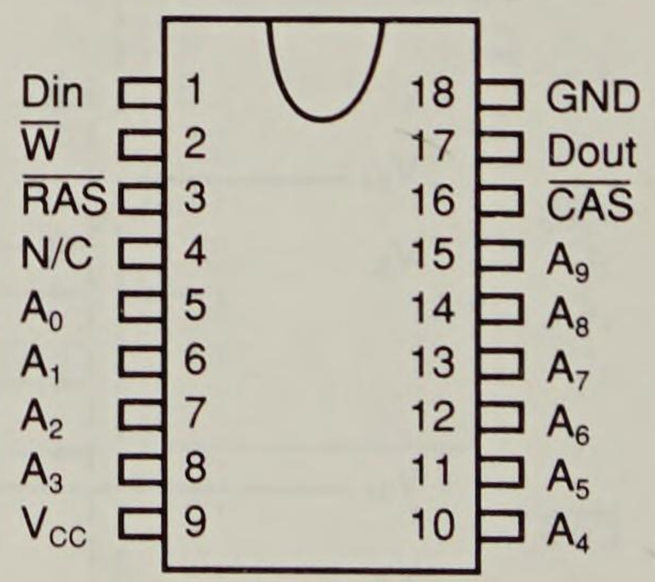



**FIGURE 10–10** The 1M × 1 DRAM.




PIN FUNCTIONS

| | |
|---|---|
| $A_0 - A_9$ | Addresses |
| Din | Data in |
| Dout | Data out |
| $\overline{CAS}$ | Column Address Strobe |
| $\overline{RAS}$ | Row Address Strobe |
| W | Write enable |
| $V_{CC}$ | +5V Supply |
| GND | Ground |

plexers. When $\overline{RAS}$ is a logic 1, the B inputs are connected to the Y outputs of the multiplexers, and when $\overline{RAS}$ input is a logic 0, the A inputs connect to the Y outputs. Because the internal row address latch is edge-triggered, it captures the row address before the address at the inputs changes to the column address. More detail on DRAM and DRAM interfacing is provided in Section 10–6.

As with the SRAM, the $\overline{W}$ pin writes data to the DRAM and the $\overline{G}$ pin enables the output connections for a read operation. ($\overline{W}$ replaces $\overline{WE}$, and $\overline{G}$ replaces $\overline{OE}$.) Figure 10–10 shows the pinout of the Texas Instruments TMX4C1024 dynamic RAM. This device is organized as a 1M × 1 memory, requiring as little as 60 ns to access data. More recently, a larger DRAM is available that is organized as a 4M × 1. A 16M × 1 and 256M × 1 are in the planning stages.

## 10–2 ADDRESS DECODING

In order to attach a memory device to the microprocessor, it is necessary to decode the address from the microprocessor to make the memory function at a unique section or partition of the memory map. Without an address decoder, only one memory device can be connected to a microprocessor, which would make it virtually useless. In this section, we describe a few of the more common address decoding techniques, as well as the decoders that are found in many systems.

### Why Decode Memory?

When a microprocessor, such as the 80386SX, is compared to the 2716 EPROM there is a difference in the number of address connection surfaces—the EPROM has 11 address connections and the 80386SX microprocessor has a 24-bit address. The microprocessor sends out a memory address whenever it reads or writes data. Since the EPROM has only 11 address inputs, there is a mismatch that must somehow be corrected. If only 11 of the microprocessor's address pins are connected to the memory, then the microprocessor will see only 2K bytes of memory instead of the 16M bytes that it expects the memory to contain. The decoder is used to match the microprocessor with the memory component.

### Simple NAND Gate Decoder

Suppose that a memory is connected to an 8-bit microprocessor that has 20 address connections. Such a device is the older 8088 microprocessor, which addresses 1 M byte of memory. Address connections $A_{10}$–$A_0$ of the 8088 are connected to address inputs $A_{10}$–$A_0$ of the EPROM, and the remaining nine address pins of the 8088 ($A_{19}$–$A_{11}$) are connected to the inputs of a NAND gate decoder (see Figure 10–11). The decoder selects the EPROM from one of the many 2K byte sections of the entire 1M byte address range of the 8088 microprocessor.

In this circuit, a single NAND gate decodes the memory address. The output of the NAND gate is a logic 0 whenever the 8088 address pins attached to its inputs ($A_{19}$–$A_{11}$) are all logic 1s. The active low, logic 0 output of the NAND gate decoder is connected to

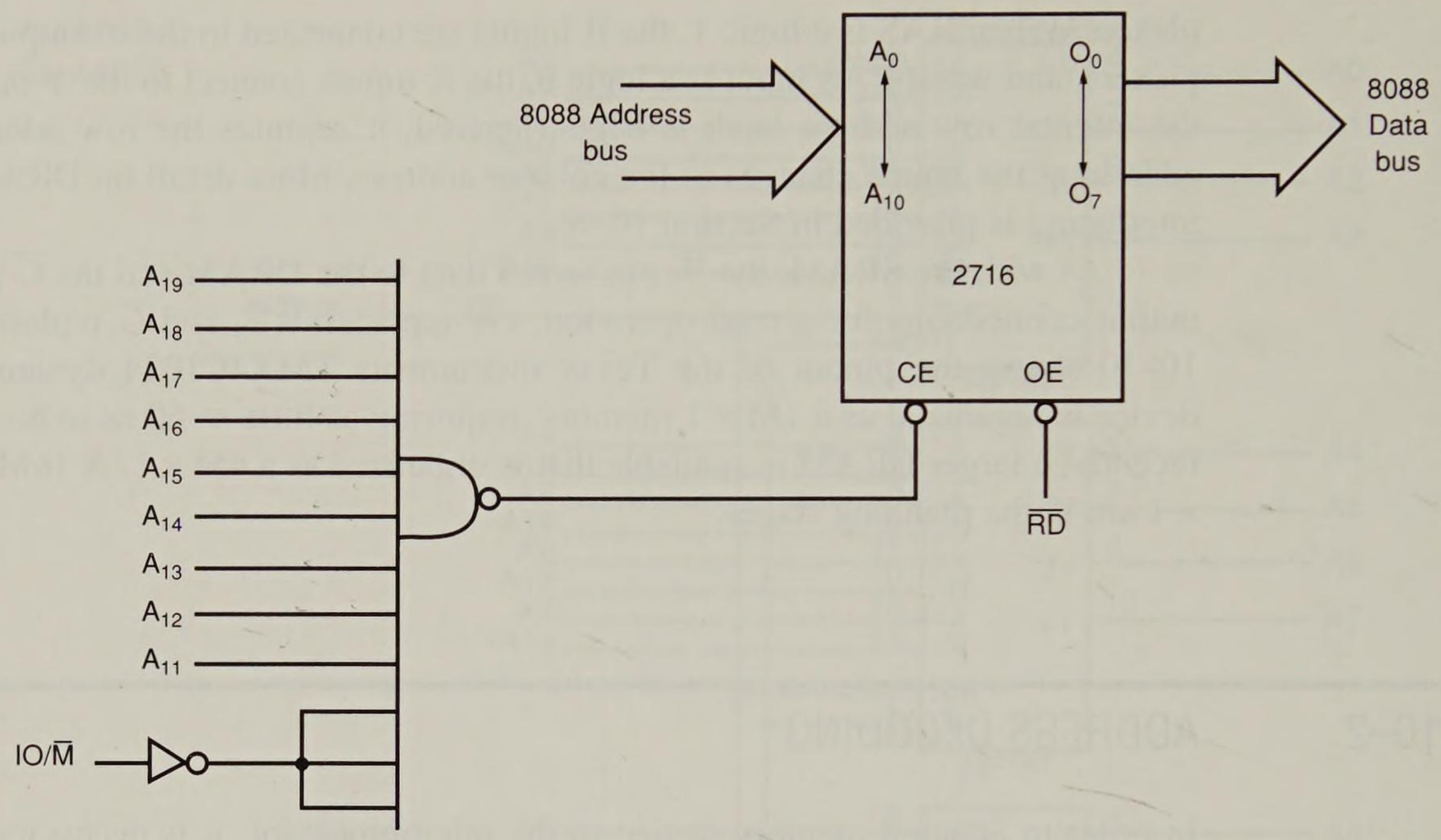



**FIGURE 10–11** A simple NAND gate decoder used to select a 2716 EPROM memory component for memory locations FF800H–FFFFFH.

the $\overline{CE}$ input, which selects (*enables*) the EPROM. Recall that whenever $\overline{CE}$ is a logic 0, data will be read from the EPROM only if $\overline{OE}$ is also a logic 0. The $\overline{OE}$ pin is activated by the 8088 $\overline{RD}$ signal or the $\overline{MRDC}$ (memory read control) signal of other family members.

If the 20-bit binary address decoded by the NAND gate is written so the left-most 9 bits are 1s and the right-most 11 bits are don't cares (X), the actual address range of the EPROM can be determined. (A don't care is a logic 1 or a logic 0, whichever is appropriate.)

**EXAMPLE 10–1**

```
1111 1111 1XXX XXXX XXXX

          or

1111 1111 1000 0000 0000 = FF800H

          to

1111 1111 1111 1111 1111 = FFFFFH
```

Example 10–1 illustrates how the address range for this EPROM is determined by writing down the externally decoded address bits ($A_{19}$–$A_{11}$) and the address bits decided by the EPROM ($A_{10}$–$A_0$) as don't cares. As the example illustrates, the don't cares are first written as 0's to locate the lowest address and then as 1s to find the highest address. Example 10–1 also shows these binary boundaries as hexadecimal addresses. Here the 2K EPROM is decoded at memory address locations FF800H–FFFFFH. Notice that this 2K

byte section of the memory is located at the reset location for the 8088, the most likely place for an EPROM.

Although this example serves to illustrate decoding, NAND gates are rarely used to decode memory, because each memory device requires its own NAND gate decoder. Because of the excessive cost of the NAND gate decoder and limited number of inputs, an alternate option must be found.

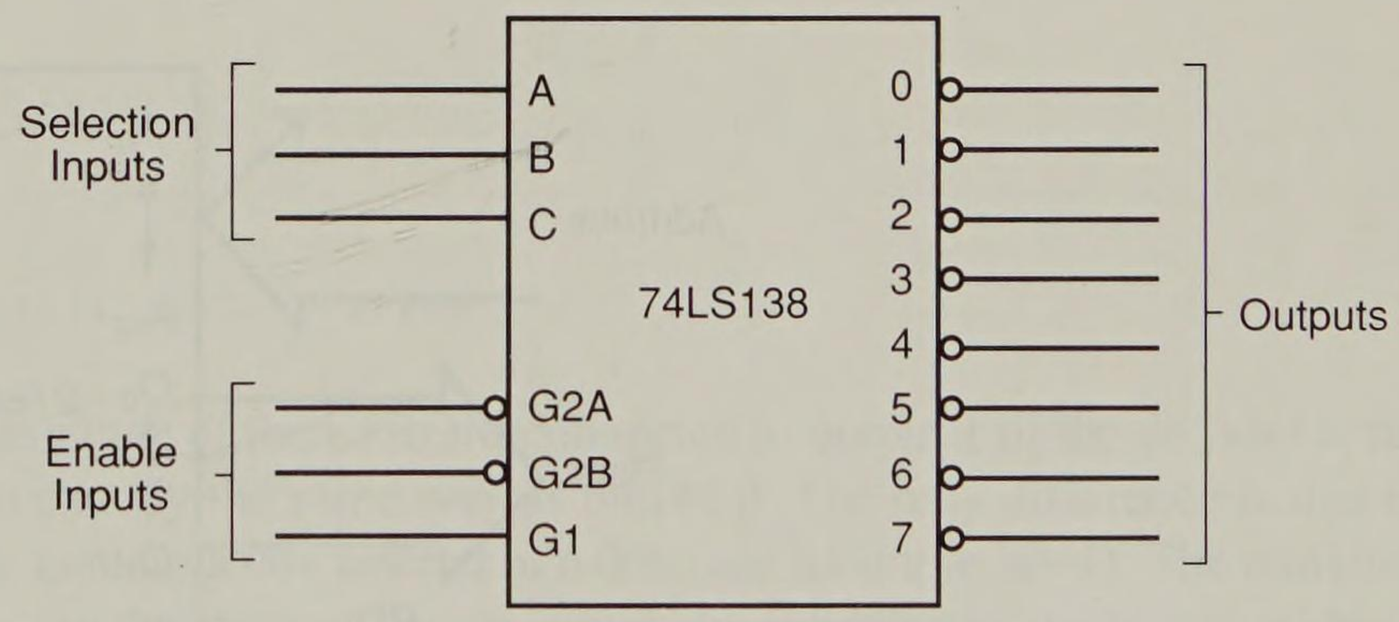
## The 3-to-8 Line Decoder (74LS138)

Another common decoder is the integrated circuit decoder found in many microprocessor-based systems. One of the more common is the 74LS138 3-to-8 line decoder. Figure 10–12 illustrates this decoder and its truth table.

The truth table shows that only one of the eight outputs ever goes low at any time. For any of the decoder's outputs to go low, the three enable inputs ($\overline{G2A}$, $\overline{G2B}$, and G1) must all be active. To be active, the $\overline{G2A}$ and $\overline{G2B}$ inputs must both be low (logic 0), and G1 must be high (logic 1).

Once the 74LS138 is enabled, the address inputs (C, B, and A) select which output pin goes low. Imagine eight EPROM $\overline{CE}$ inputs connected to the eight outputs of the

**FIGURE 10–12** The 74LS138 3-to-8 line decoder and function table.




| Inputs | | | | | | Outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Enable | | | Select | | | | | | | | | | |
| $\overline{G2A}$ | $\overline{G2B}$ | G1 | C | B | A | $\overline{0}$ | $\overline{1}$ | $\overline{2}$ | $\overline{3}$ | $\overline{4}$ | $\overline{5}$ | $\overline{6}$ | $\overline{7}$ |
| 1 | X | X | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| X | 1 | X | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| X | X | 0 | X | X | X | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

decoder! This is a very powerful device because it selects eight different memory devices at the same time.

*Sample Decoder Circuit.* Notice that the outputs of the decoder illustrated in Figure 10–13 are connected to eight different 2764 EPROM memory devices. Here the decoder selects eight 8K byte blocks of memory for a total of 64K bytes of memory. This figure also illustrates the address range of each memory device and the common connections to the memory devices. Notice that all the address connections from the 8088 are connected to this circuit. Also notice that the decoder's outputs are connected to the $\overline{CE}$ inputs of the EPROMs, and the $\overline{RD}$ signal from the 8088 is connected to the $\overline{OE}$ inputs of the EPROMs. This allows only the selected EPROM to be enabled and to send its data to the microprocessor through the data bus whenever $\overline{RD}$ becomes a logic 0.

In this circuit, a 3-input NAND gate is connected to address bits $A_{19}$–$A_{17}$. When all three address inputs are high, the output of this NAND gate goes low and enables input $\overline{G2B}$ of the 74LS138. Input G1 is connected directly to $A_{16}$. In other words, in order to enable this decoder, the first four address connections ($A_{19}$–$A_{16}$) must all be high.

The address inputs C, B, and A connect to microprocessor address pins $A_{15}$–$A_{13}$. These three address inputs determine which output pin goes low and which EPROM is selected, whenever the 8088 outputs a memory address within this range to the memory system.

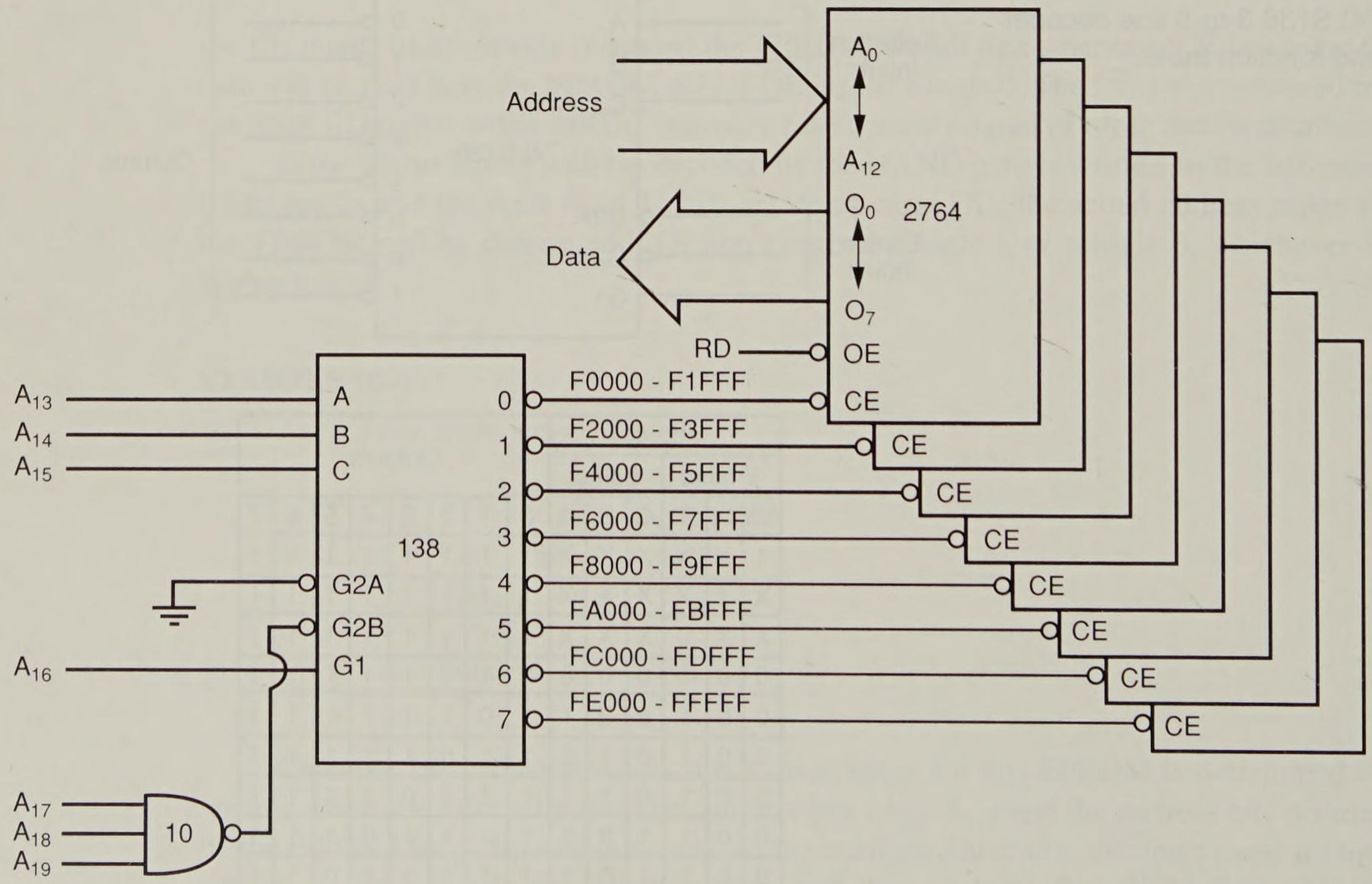



**FIGURE 10–13** A circuit that uses eight 2764 EPROMs for a 64K × 8 section of memory in an 8088 microprocessor-based system. The addresses selected in this circuit are F0000H–FFFFFH.

**EXAMPLE 10–2**

```
1111 XXXX XXXX XXXX XXXX

          or

1111 0000 0000 0000 0000 = F0000H

          to

1111 1111 1111 1111 1111 = FFFFH
```

Example 10–2 shows how the address range of the entire decoder is determined. Notice that the range is location F0000H–FFFFFH. This is a 64K byte span of the memory.

How is it possible to determine the address range of each memory device attached to the decoder's outputs? Again, the binary bit pattern is written down; this time the C, B, and A address inputs are *not* don't cares. Example 10–3 shows how output 0 of the decoder is made to go low to select the EPROM attached to that pin. Here C, B, and A are shown as logic 0s.

**EXAMPLE 10–3**

```
     CBA
1111 0000 XXXX XXXX XXXX

          or

1111 0000 0000 0000 0000 = F0000H

          to

1111 0001 1111 1111 1111 = F1FFFH
```

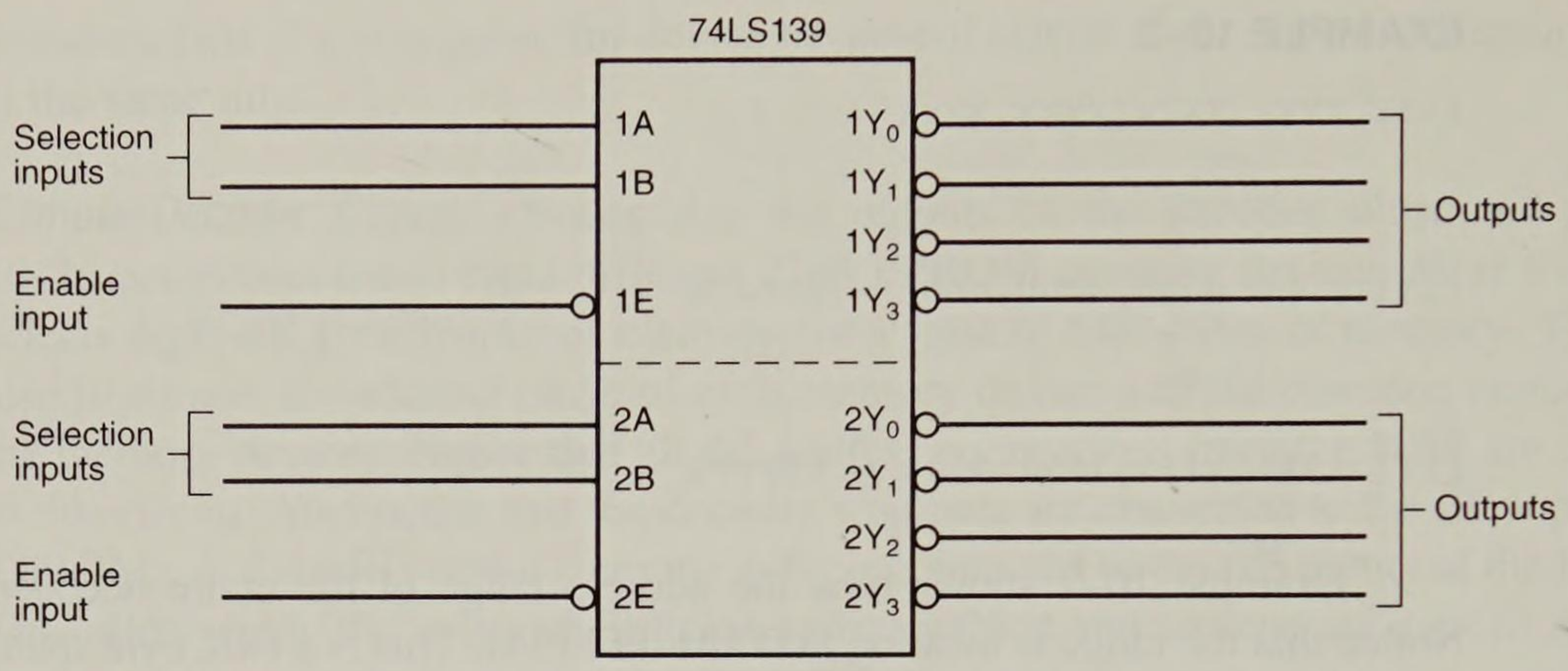
If the address range of the EPROM connected to output 1 of the decoder is required, it is determined in exactly the same way as output 0. The only difference is that now the C, B, and A inputs contain a 001 instead of a 000 (see Example 10–4). The remaining output address ranges are determined the same way, by substituting the binary address of the output pin into C, B, and A.

**EXAMPLE 10–4**

```
     CBA
1111 001X XXXX XXXX XXXX

          or

1111 0010 0000 0000 0000 = F2000H

          to

1111 0011 1111 1111 1111 = F3FFFH
```

## The Dual 2-to-4 Line Decoder (74LS139)

Another decoder that finds some application is the 74LS139 dual 2-to-4 line decoder. Figure 10–14 illustrates both the pinout and the truth table for this decoder. The 74LS139




| Inputs | | | Outputs | | | |
|---|---|---|---|---|---|---|
| $\overline{E}$ | A | $\overline{B}$ | $\overline{Y_0}$ | $\overline{Y_1}$ | $\overline{Y_2}$ | $\overline{Y_3}$ |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | X | X | 1 | 1 | 1 | 1 |

**FIGURE 10–14** The pinout and truth table of the 74LS139 dual 2-to-4 line decoder.

contains two separate 2-to-4 line decoders, each with its own address, enable, and output connections.

## PROM ADDRESS DECODER

Another common address decoder is the bipolar PROM, used because of its larger number of input connections that reduce the number of other circuits required in a system memory address decoder. The 74LS138 decoder has six inputs used for address connections. The PROM decoder may have many more inputs for address decoding.

For example, a 82S147 (512 × 8) PROM used as an address decoder has 10 input connections and 8 output connections. It can replace the circuit illustrated in Figure 10–13 without the extra 3-input NAND gate. This saves space on the printed circuit board and reduces the cost of a system.

Figure 10–15 illustrates this address decoder with the PROM in place. The PROM is a memory device that must be programmed with the correct binary bit pattern to select the eight EPROM memory devices. The PROM itself has 9 address inputs that select one of the 512 internal 8-bit memory locations. The remaining input ($\overline{CE}$) must be grounded

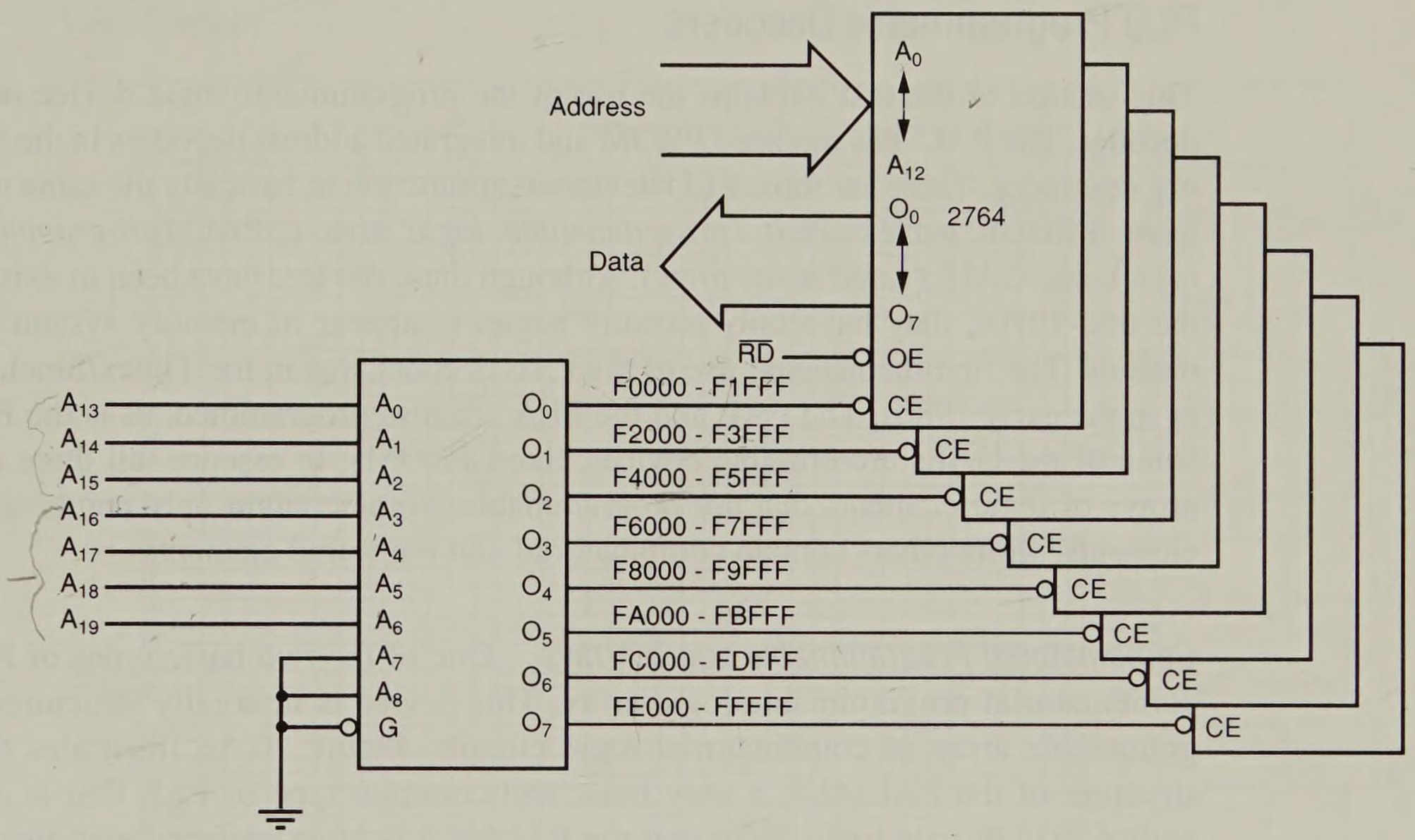



**FIGURE 10–15** A memory system using the TPB28L42, 512 × 8 PROM as an address.

**TABLE 10–1** The 82S147 PROM programming pattern for the circuit in Figure 10–15

| *Inputs* | | | | | | | | | | *Outputs* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *$\overline{OE}$* | *A8* | *A7* | *A6* | *A5* | *A4* | *A3* | *A2* | *A1* | *A0* | *00* | *01* | *02* | *03* | *04* | *05* | *06* | *07* |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| all other address combinations | | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

because if this PROM's outputs float to their high-impedance state, then one or more of the EPROMs might be selected by noise impulses in the system.

Table 10–1 illustrates the binary bit pattern programmed into each PROM location in order to select the eight different EPROMs. The main advantage to using a PROM is that the address map is easily changed in the field. Because the PROM comes with all the locations programmed as logic 1s, only 8 of the 512 locations must be programmed. This saves valuable time for the manufacturer.

## PLD Programmable Decoders

This section of the text explains the use of the programmable logic device or PLD as a decoder. The PAL* has replaced PROM and integrated address decoders in the latest memory interfaces. There are three PLD devices that function in basically the same manner, but have different names: PLA (*programmable logic array*), PAL (*programmable array logic*), and GAL† (*gated array logic*). Although these devices have been in existence since the mid-1970s, they have only recently begun to appear in memory system and digital designs. The first mainstream use of the PAL decoder was in the Timex/Sinclair computer in the early 1980s. The PAL and the PLA are fuse programmed, as is the PROM, and some of the GALs are erasable devices, like EPROMs. In essence, all three devices are arrays of logic elements that are programmable. Some contain only combinatorial logic elements, while others contain combinatorial and registered elements.

*Combinatorial Programmable Logic Arrays.* One of the two basic types of PALs is the combinatorial programmable logic array. This device is internally structured as a programmable array of combinatorial logic circuits. Figure 10–16 illustrates the internal structure of the PAL16L8, a very basic and common type of PAL that is constructed with AND/OR gate logic. Note that the PAL16L8 is often replaced with the GAL16V8 in many new designs, because it can function as the PAL16L8 or PAL16R8. The PAL16L8 has 10 fixed inputs, 2 fixed outputs, and 6 pins that are programmable as inputs or outputs. Each output pin is generated from a 7-input OR gate that has an AND gate attached to each input. The outputs of the OR gates pass through a three-state inverter that defines each out as an AND/NOR function. Before being programmed, fuses connect all of the vertical and horizontal connections illustrated in Figure 10–16. Programming is accomplished by blowing fuses to connect various inputs to the OR gate array through wired-AND gates. The wired-AND function is performed at each input connection, to allow a product term of up to 16 inputs. A logic expression using the PAL16L8 can have 7 product terms with 16 inputs NORed together to generate an output expression. This device is ideal as a memory address decoder because of its structure. It is also ideal because its outputs are active low, and most memory devices contain an active low select input.

Fortunately we don't have to choose the fuses by number for programming. We program the PAL using a software package called PALASM‡ (the PAL assembler). The PALASM program and its syntax are an industry standard for programming PAL devices. Example 10–5 shows a program that decodes the same areas of memory as decoded in Figure 10–15. Note that this program was developed using a text editor such as EDIT, available with Microsoft DOS version 5.0/6.X. It can also be developed using an editor that comes with the PALASM package. Various editors attempt to ease the task of defining the pins, but we believe it is easier to use EDIT and the listing as shown.

---

*PAL is a registered trademark of Monolithic Memories, Inc.

†GAL is a registered trademark of LATTICE Semiconductors, Inc.

‡PALASM is a registered trademark of Monolithic Memories, Inc.

Logic Diagram

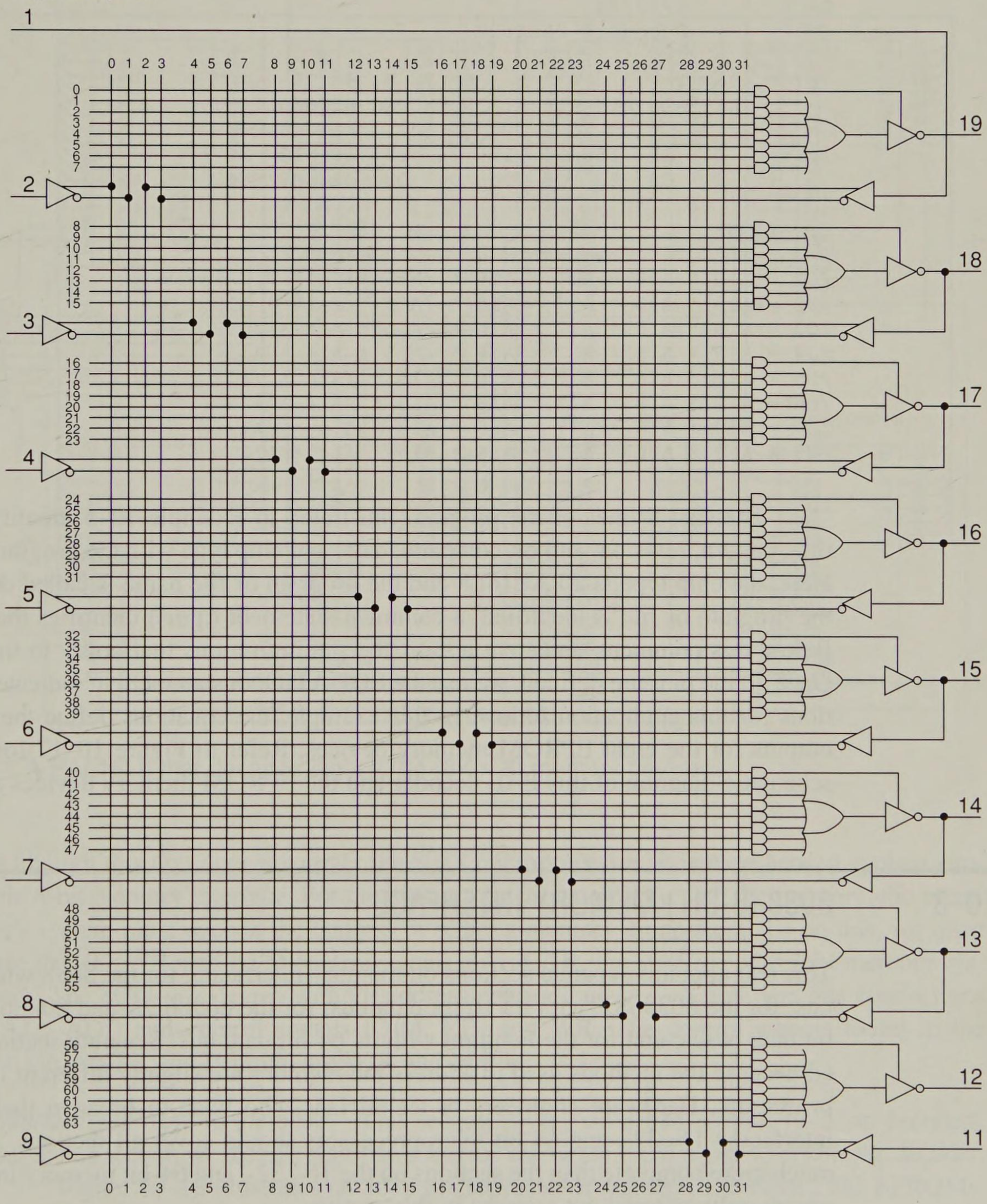



**FIGURE 10–16** The PAL 16L8. (Copyright Advanced Micro Devices, Inc., 1988. Reprinted with permission of copyright owner. All rights reserved.)

## EXAMPLE 10–5

```
TITLE       Address Decoder
PATTERN     Test 1
REVISION    A
AUTHOR      Barry B. Brey
```

```
COMPANY     Symbiotic Systems
DATE        6/23/93
CHIP        Decoder1 PAL16L8

;pins  1    2    3    4    5    6  7  8  9  10
       A19  A18  A17  A16  A13  NC NC NC NC GND

;pins   11   12   13   14   15   16   17   18   19   20
        NC   O8   O7   O6   O5   O4   O3   O2   O1   VCC

EQUATIONS

/O1 = A19 * A18 * A17 * A16 * /A15 * /A14 * /A13
/O2 = A19 * A18 * A17 * A16 * /A15 * /A14 * A13
/O3 = A19 * A18 * A17 * A16 * /A15 * A14 * /A13
/O4 = A19 * A18 * A17 * A16 * /A15 * A14 * A13
/O5 = A19 * A18 * A17 * A16 * A15 * /A14 * /A13
/O6 = A19 * A18 * A17 * A16 * A15 * /A14 * A13
/O7 = A19 * A18 * A17 * A16 * A15 * A14 * /A13
/O8 = A19 * A18 * A17 * A16 * A15 * A14 * A13
```

The first 8 lines of the program illustrated in Example 10–5 identify the program title, pattern, revision, author, company, date, and chip type with the program or file name. Here, the chip type is a PAL16L8 and the program or file name is called decoder1. After the program or file is identified, a comment statement (;pins) identifies the pin numbers. Below this comment statement appear the pin definitions that apply to this application. Once all the pins are defined, we use the EQUATIONS statement to indicate that the equations for this application follow. In this example, the equations define the 8-chip enable outputs for the eight EPROM memory devices. Refer to Figure 10–17 for the complete schematic diagram of this PAL decoder and the EPROM memory devices attached to it.

## 10–3 8088 (8-BIT) MEMORY INTERFACE

This text contains separate sections on memory interfacing for the 8088 with its 8-bit data bus; for the 80386SX with its 16-bit data bus; for the 80386DX and 80486 with their 32-bit data buses; and for the Pentium with its 64-bit data bus. Separate sections are provided because the methods used to address the memory are slightly different in these microprocessors. Hardware engineers or technicians who wish to broaden their expertise in interfacing 16- 32-, and 64-bit microprocessors should cover all sections. This section is much more complete than the sections on the 16-, 32-, and 64-bit memory interface, which covers only material not covered in this section.

In this section, we examine the memory interface to both RAM and ROM and explain parity checking, which is commonplace in many microprocessor-based computer systems. We also briefly mention error correction schemes currently available to memory system designers.

### Basic 8-Bit Memory Interface

The 8088 microprocessor has an 8-bit data bus, which makes it ideal to connect to the common 8-bit memory devices available today. It also makes it ideal as a simple controller.

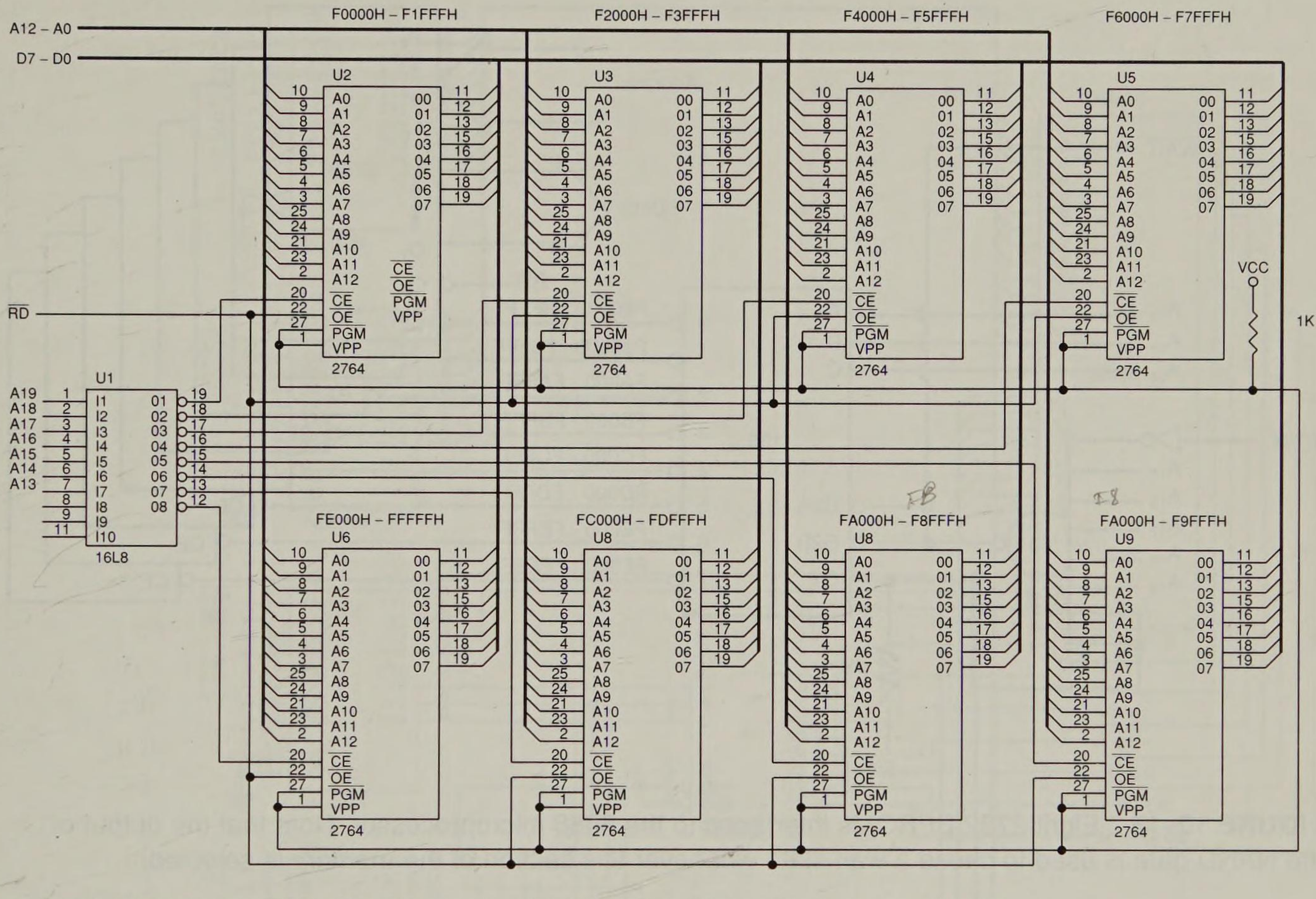



**FIGURE 10–17** A PAL16L8 that decodes eight 2764 (8K × 8) memory devices.

Although this text does not cover the 8088 microprocessor, it is used here to explain simple 8-bit memory interface. For the 8088 to function correctly with the memory, the memory system must decode the address to select a memory component. To do this, we must use the $\overline{RD}$, $\overline{WR}$, and IO/$\overline{M}$ control signals provided by the 8088 to control the memory system. The 8088 is a device with 20 address connections ($A_{19}$–$A_0$), 8 data bus connections ($AD_7$–$AD_0$) and control signals IO/$\overline{M}$, $\overline{RD}$, and $\overline{WR}$. The control signals found in the 80386, 80486, and Pentium are W/$\overline{R}$ and M/$\overline{IO}$.

*Interfacing EPROM to the 8088.* This section is very similar to Section 10–2, on decoders, but in this section we discuss wait states and the use of the IO/M signal to enable the decoder.

Figure 10–18 illustrates an 8088 microprocessor connected to eight 2732 EPROMs, 4K × 8 memory devices that are in very common use today. The 2732 has one more address input ($A_{11}$) than the 2716 and twice the memory. The device in this illustration decodes eight 4K × 8 blocks of memory, for a total of 32K × 8 bits of physical address space for the 8088.

The decoder (74LS138) is connected a little differently than might be expected because the slower version of this type of EPROM has a memory access time of 450 ns. When the 8088 is operated with a 5 MHz clock, it allows 460 ns for the memory to access data. Because of the decoder's added time delay (12 ns), it is impossible for this memory to function within 460 ns. In order to correct this problem, we must add a NAND gate to

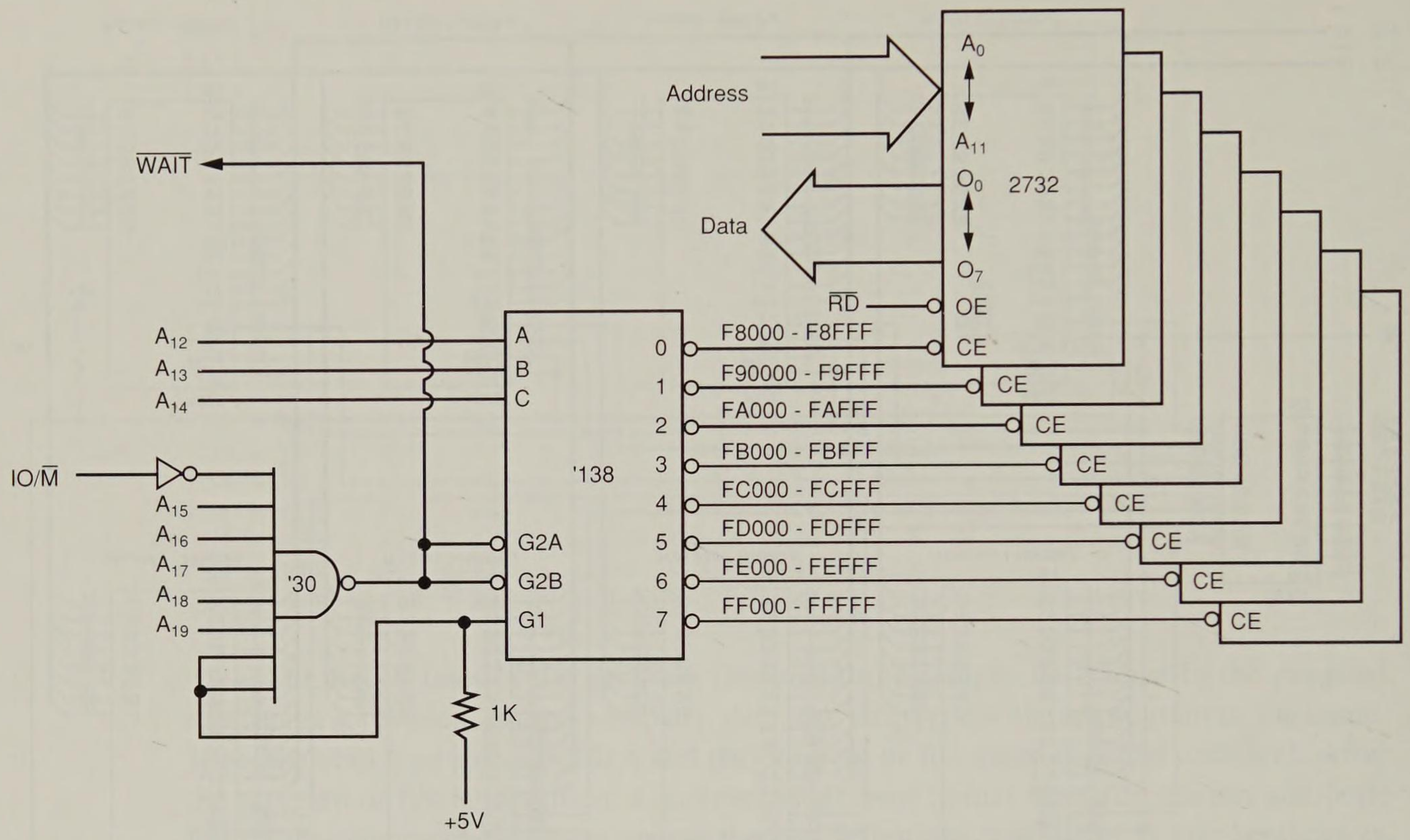



**FIGURE 10–18** Eight 2732 EPROMs interfaced to the 8088 microprocessor. Note that the output of the NAND gate is used to cause a wait state whenever this section of the memory is selected.

generate a signal to enable the decoder and a signal for the wait state generator. With a wait state inserted every time this section of the memory is accessed, the 8088 will allow 660 ns for the EPROM to access data. Recall that an extra wait state adds 200 ns (1 clock) to the access time. 660 ns is ample time for a 450 ns memory component to access data, even with the delays introduced by the decoder and any buffers added to the data bus.

Notice that the decoder is selected for a memory address range that begins at location F0000H and continues through location FFFFFH—the upper 64K bytes of memory. This section of memory is an EPROM, because FFFF0H is where the 8088 starts to execute instructions after a hardware reset. We often call location FFFF0H the *cold start* location. The software stored in this section of memory contains a JMP instruction at location FFFF0H that jumps to location F0000H so the remainder of the program can execute. Note that in a personal computer, the BIOS (basic I/O system) ROM begins at location F0000H, and a jump to F0000H appears at location FFFF0H.

*Interfacing RAM to the 8088.* RAM is a little easier to interface than EPROM because most RAM memory components do not require wait states. An ideal section of the memory for the RAM is the very bottom, which contains vectors for interrupts. Interrupt vectors (discussed in more detail in Chapter 12) are often modified by software packages, so it is rather important to encode this section of the memory with RAM.

In Figure 10–19, sixteen 62256 32K × 8 static RAMS are interfaced to the 8088, beginning at memory location 00000H. This circuit board uses two decoders to select the

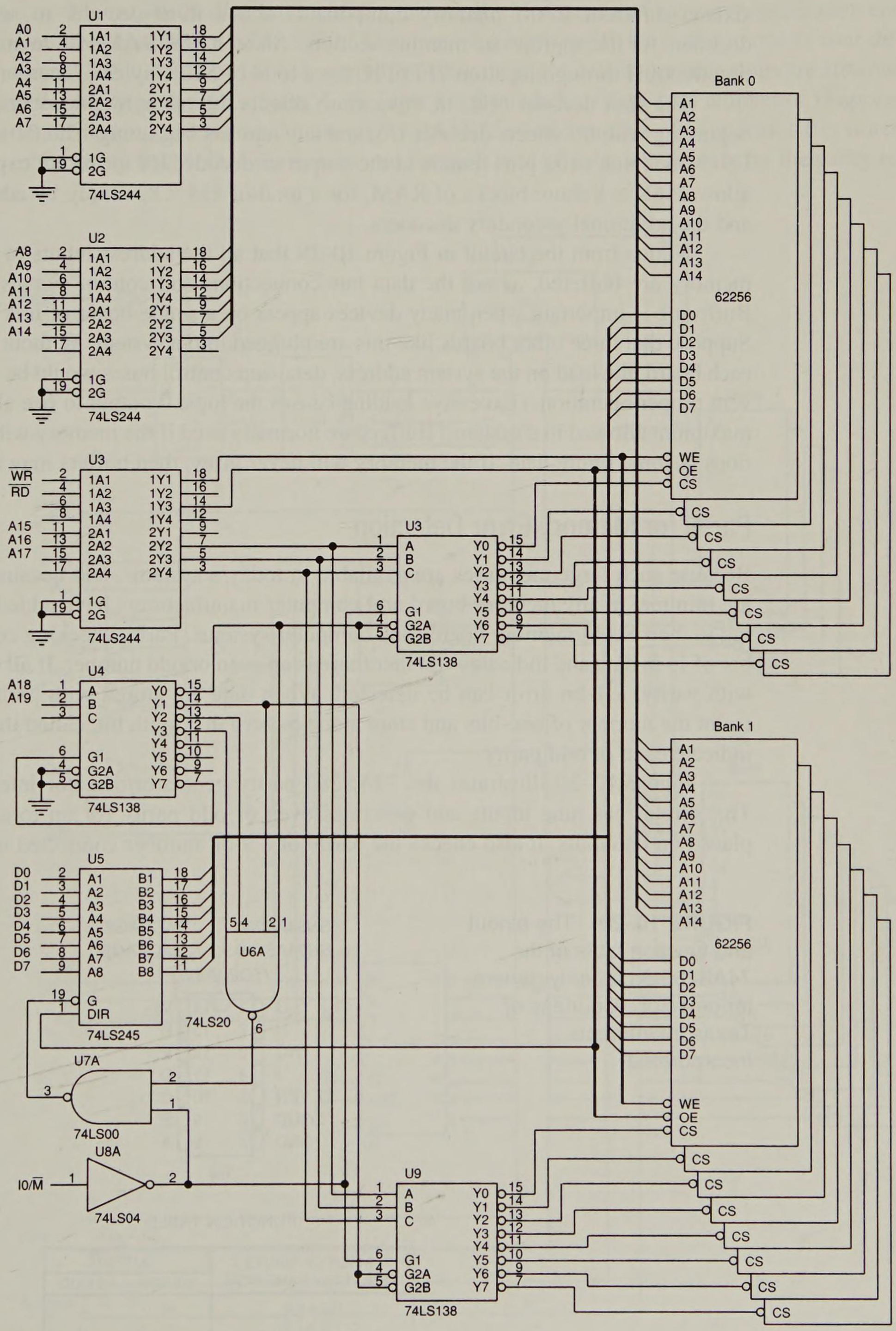



**FIGURE 10–19** A 512K byte static memory system using sixteen 62255 SRAMs.

sixteen different RAM memory components and a third decoder to select the other decoders for the appropriate memory sections. Sixteen 32K RAMs fill memory from location 00000H through location 7FFFFH, for a total of 512K bytes of memory.

The first decoder (U4) in this circuit selects the other two decoders. Any address beginning with 00 selects decoder U3, and any address beginning with 01 selects decoder U9. Notice that extra pins remain at the output of decoder U4 for future expansion. These allow 256K × 8 more blocks of RAM, for a total of 1M × 8, simply by adding the RAM and the additional secondary decoders.

Notice from the circuit in Figure 10–19 that all the address inputs to this section of memory are buffered, as are the data bus connections and control signals $\overline{RD}$ and $\overline{WR}$. Buffering is important when many devices appear on a single board or in a single system. Suppose that three other boards like this are plugged into a system. Without the buffers on each board, the load on the system address, data, and control buses would be enough to prevent proper operation. (Excessive loading causes the logic 0 output to rise above the 0.8 V maximum allowed in a system.) Buffers are normally used if the memory will contain additions at some future date. If the memory will never grow, then buffers may not be needed.

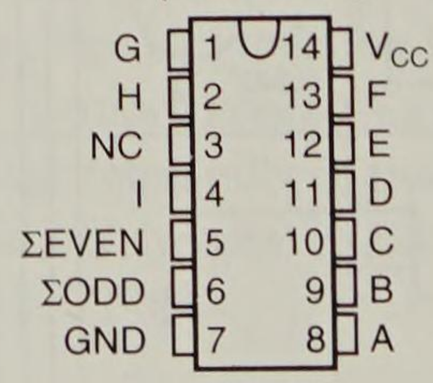
## Parity for Memory Error Detection

Because such large memories are available in today's systems, and because circuit costs are minimal, many memory board and computer manufacturers have added parity checking to their RAM memory boards and computer systems. Parity checking counts the number of 1s in data and indicates whether there is an even or odd number. If all data are stored with parity, a 1-bit error can be detected. When data are stored with parity, we usually count the number of one-bits and store a one or zero in a ninth bit, called the *parity bit*, to indicate even or odd parity.

Figure 10–20 illustrates the 74AS280 parity generator/detector integrated circuit. This circuit has nine inputs and generates even or odd parity for up to a 9-bit number placed on its inputs. It also checks the parity of a 9-bit number connected to its inputs.

**FIGURE 10–20** The pinout and function table of the 74AS280 9-bit parity generator/detector. (Courtesy of Texas Instruments Incorporated)

SN54AS280 . . . J PACKAGE
SN74AS280 . . . N PACKAGE
(TOP VIEW)




(a)

FUNCTION TABLE

| NUMBER OF INPUTS A THRU 1 THAT ARE HIGH | OUTPUTS | |
|---|---|---|
| | Σ EVEN | Σ ODD |
| 0,2,4,6,8 | H | L |
| 1,3,5,7,9 | L | H |

(b)

Figure 10–21 illustrates a 16K × 8 static RAM system that has parity generation and detection. Notice that a 74AS280 (A) generates a ninth parity bit stored in one of four different TMS4044 4K × 1 RAM memories. Here the eight data bus connections are attached to the parity generator's inputs A–H. Input I is grounded, so if an even number of 1s appear on the data bus, a 1 is stored in the parity RAM. If an odd number of 1s appear, a 0 is stored in the parity RAM. Odd parity is stored for each byte of data, which includes the parity bit

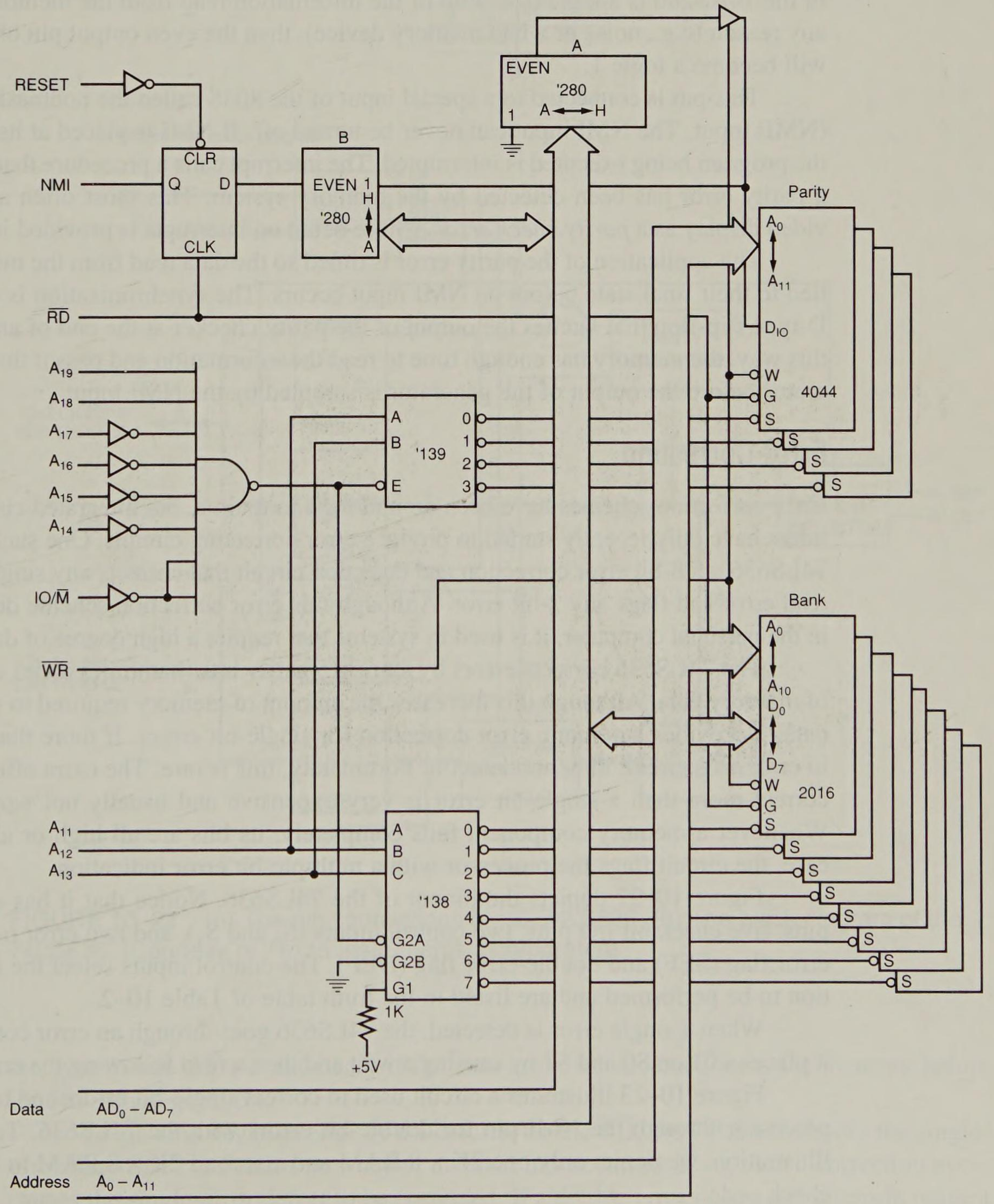



**FIGURE 10–21** A 16K × 8 memory system that contains a parity error detection circuit.

written to the memory. For example, if the data bus contains a 11000000, the parity generated by the 74AS280 is a logic 1. This means that the 9-bit number stored for this even data is odd.

When data are read from the memory, each datum is connected to another 74AS280 (B) to check its parity. In this case, all nine inputs to the checker are connected. Inputs A–H are connected to the data RAM's outputs, and input I, the ninth input, is connected to the parity RAM. If parity is odd, as it is if everything is correct, the even parity output of the 74AS280 is a logic 0. If a bit of the information read from the memory changes for any reason (e.g., noise or a bad memory device), then the even output pin of the 74AS280 will become a logic 1.

This pin is connected to a special input of the 8088 called the nonmaskable interrupt (NMI) input. The NMI input can never be turned off. If NMI is placed at its logic 1 level, the program being executed is interrupted. The interrupt calls a procedure that indicates that a parity error has been detected by the memory system. This most often appears on the video display as a *parity check error*. (More detail on interrupts is provided in Chapter 12.)

The application of the parity error is timed so the data read from the memory are settled to their final state before an NMI input occurs. The synchronization is provided by a D-type flip-flop that latches the output of the parity checker at the end of an $\overline{RD}$ cycle. In this way, the memory has enough time to read the information and pass it through the generator before the output of the generator is sampled by the NMI input.

## Error Correction

Error correction schemes have been around for a long time, but integrated circuit manufacturers have only recently started to produce error correcting circuits. One such circuit is the 74LS636, an 8-bit error correction and detection circuit that corrects any single-bit memory read error and flags any 2-bit error. Although this error correction scheme does not appear in the personal computer, it is used in systems that require a high degree of data integrity.

The 74LS636 corrects errors by storing 5 parity bits (hamming code) with each byte of memory data. Although this increases the amount of memory required to store a byte of data, it provides automatic error correction for single-bit errors. If more than two bits are in error, this circuit may not detect it. Fortunately, this is rare. The extra effort required to correct more than a single-bit error is very expensive and usually not worth the effort. Whenever a memory component fails completely, its bits are all high or all low. In this case, the circuit flags the processor with a multiple-bit error indication.

Figure 10–22 depicts the pinout of the 74LS636. Notice that it has eight data I/O pins, five check bit I/O pins, two control inputs ($S_0$ and $S_1$), and two error outputs: single-error flag (SEF) and double-error flag (DEF). The control inputs select the type of operation to be performed and are listed in the truth table of Table 10–2.

When a single error is detected, the 74LS636 goes through an error correction cycle: it places a 01 on S0 and S1 by causing a wait and then a read following the error correction.

Figure 10–23 illustrates a circuit used to correct single-bit errors and to interrupt the processor through the NMI pin for double-bit errors with the 74LS636. To simplify the illustration, we depict only one 2K × 8 RAM and a second 2K × 8 RAM to store the 5-bit check code.

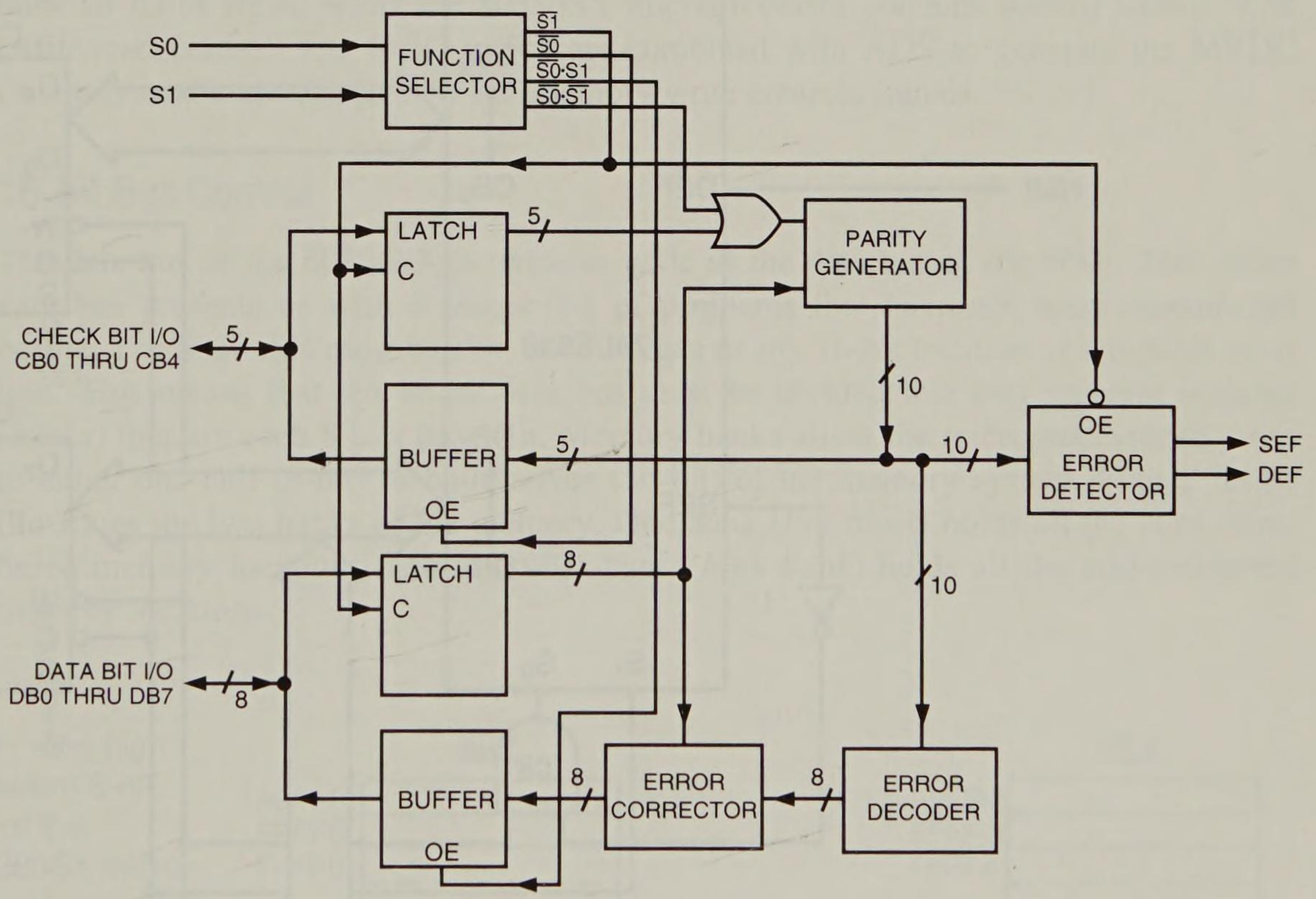
The connection of this memory component is different from that of the previous example. Notice that the $\overline{S}$ or $\overline{CS}$ pin is grounded, and data bus buffers control the flow to

Pin assignments

| J, N PACKAGES | | | |
|---|---|---|---|
| 1 | DEF | 11 | CB4 |
| 2 | DB0 | 12 | nc |
| 3 | DB1 | 13 | CB3 |
| 4 | DB2 | 14 | CB2 |
| 5 | DB3 | 15 | CB1 |
| 6 | DB4 | 16 | CB0 |
| 7 | DB5 | 17 | S0 |
| 8 | DB6 | 18 | S1 |
| 9 | DB7 | 19 | SEF |
| 10 | GND | 20 | $V_{CC}$ |

(a)

Functional block diagram




(b)

**FIGURE 10–22** (a) The pin connections of the 74LS636. (b) The block diagram of the 74LS636. (Courtesy of Texas Instruments Incorporated)

the system bus. This is necessary if the data are to be accessed from the memory before the $\overline{RD}$ strobe goes low.

On the next negative edge of the clock after the $\overline{RD}$, the 74LS636 checks the single-error flag (SEF) to determine whether an error has occurred. If so, then a correction cycle causes the single-error defect to be corrected. If a double error occurs, an interrupt request is generated by the double-error flag (DEF) output, which is connected to the NMI pin of the microprocessor.

**TABLE 10–2** Control bits $S_0$ and $S_1$

| | | | Error Flags | |
|---|---|---|---|---|
| $S_0$ | $S_1$ | *Function* | *SEF* | *DEF* |
| 0 | 0 | Write check word | 0 | 0 |
| 0 | 1 | Correct data word | * | * |
| 1 | 0 | Read data | 0 | 0 |
| 1 | 1 | Latch data | * | * |

**Note*: These levels are determined by the type of error.

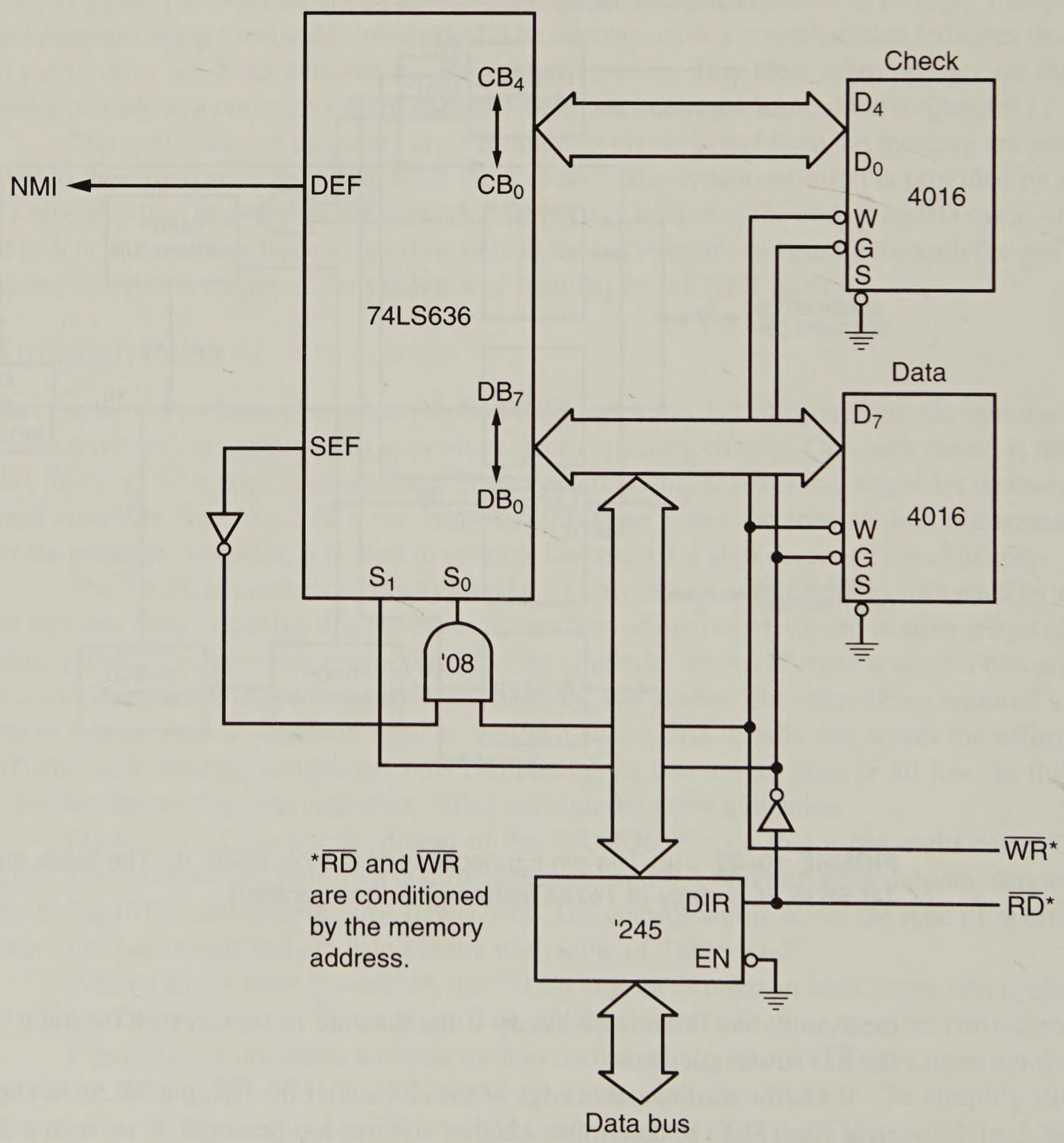



**FIGURE 10–23** An error detection and correction circuit using the 74LS636.

# 10–4 80386SX (16-BIT) MEMORY INTERFACE

The 80386SX microprocessor differs from the 8088 and its 8-bit wide memory system in two ways: (1) the data bus is 16 bits wide instead of 8 bits wide and (2) there are new control signals called *bus high enable* ($\overline{BHE}$) and *bus low enable* ($\overline{BLE}$). The address bit $A_0$ is also used differently and has been renamed bus low enable ($\overline{BLE}$). Because this section of the text is based on information provided in Section 10–3, it is extremely important that you read the previous section first.

A few other differences exist between the 8088 and 80386SX. The 80386SX contains a 24-bit address bus ($A_{23}$–$A_0$) instead of the 20-bit address bus ($A_{19}$–$A_0$) of the 8088, because the 80386SX addresses 16M bytes of memory instead of 1M bytes. The 8088 contains an IO/$\overline{M}$ signal while the 80386SX microprocessor contains control signals W/$\overline{R}$ (write/read control) and M/$\overline{IO}$ which are combined with $\overline{ADS}$ to generate the $\overline{MRDC}$ (memory read control) and $\overline{MWTC}$ (memory write control) signals.

## 16-Bit Bus Control

The data bus of the 80386SX is twice as wide as the data bus of the 8088. This wider data bus presents us with a unique set of problems that have not been encountered before. The 80386SX must be able to write data to any 16-bit location *or* any 8-bit location. This means that the 16-bit data bus must be divided into two separate sections (*banks*) that are each 8 bits in width. Memory banks allow the microprocessor to write to either one half (8-bit) or both halves (16-bit) of the memory system. Figure 10–24 illustrates the two banks of the memory. One bank (*low bank*) holds all the even-numbered memory locations, and the other bank (*high bank*) holds all the odd-numbered memory locations.

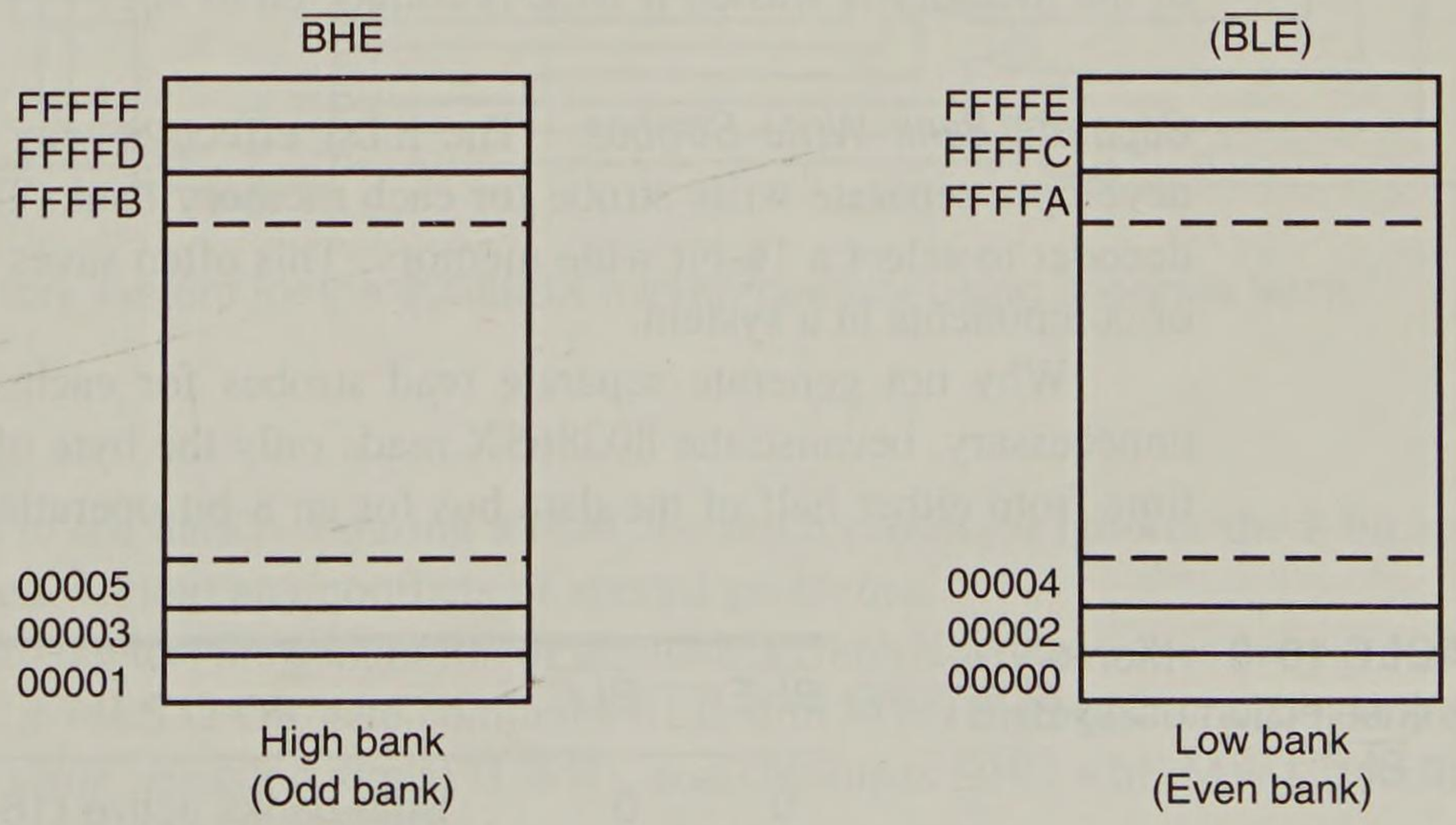



**FIGURE 10–24** The high (odd) and low (even) 8-bit memory banks of the 8086/80286/80386SX microprocessor.

Note: $A_0$ is labeled $\overline{BLE}$ (BUS low enable) on the 80386SX.

The 80386SX uses the $\overline{BHE}$ (high bank enable) and $\overline{BLE}$ (low bank enable) signals to select one or both banks of memory used for the data transfer. Table 10–3 depicts the logic levels on these two pins and the bank or banks selected.

Bank selection is accomplished in two ways: (1) a separate write signal is developed for each bank of the memory, or (2) separate decoders are used for each bank. A comparison of both techniques reveals that the first method is by far the least costly approach to memory interface for the 80386SX microprocessor, because it often requires fewer components to implement.

*Separate Bank Decoders.* The use of separate bank decoders is often the least effective way to decode memory addresses for the 80386SX. This method is sometimes used, but it is difficult to understand why. One possible reason is that only the bank that is selected performs an operation, which can reduce the power consumption of the memory system. This is important in battery-operated equipment.

Figure 10–25 illustrates two 74LS138 decoders used to select 64K RAM memory components for the 80386SX microprocessor (24-bit address). Here decoder A has the $\overline{BLE}$ pin attached to its $\overline{G2A}$ input, and decoder B has the $\overline{BHE}$ signal attached to its $\overline{G2A}$ input. Because the decoder will not activate until all its enable inputs are active, decoder A activates only for a 16-bit operation or an 8-bit operation from the low bank, and decoder B activates for a 16-bit operation or an 8-bit operation to the high bank. The two decoders and the sixteen 64K-byte RAMs they control represent the 1M range of the 80386SX memory system. Note that the range of addresses decoded in this illustration are from location 000000H through location 0FFFFFH.

Notice that the $\overline{BLE}$ address pin does not connect to the memory; instead, it connects to the decoder. Recall that $\overline{BLE}$ is actually address signal $A_0$. Also notice that address bus bit position $A_1$ is connected to memory address input $A_0$, $A_2$ is connected to $A_1$, and so forth. If $\overline{BLE}$, which represents $A_0$, would be attached to the $A_0$ address pin of memory, every other memory location in each bank of memory would be used. This means that half of the memory is wasted if $\overline{BLE}$ is connected to $A_0$.

*Separate Bank Write Strobes.* The most effective way to handle bank selection is to develop a separate write strobe for each memory bank. This technique requires only one decoder to select a 16-bit wide memory. This often saves money and reduces the number of components in a system.

Why not generate separate read strobes for each memory bank? This is usually unnecessary, because the 80386SX reads only the byte of data that it needs at any given time from either half of the data bus for an 8-bit operation. If 16-bit sections of data are

**TABLE 10–3** Memory bank selection using $\overline{BHE}$ and $\overline{BLE}$

| $\overline{BHE}$ | $\overline{BLE}$ | *Function* |
|---|---|---|
| 0 | 0 | Both banks active (16-bit transfer through $D_{15}$–$D_0$) |
| 0 | 1 | High bank active (8-bit transfer through $D_{15}$–$D_8$) |
| 1 | 0 | Low bank active (8-bit transfer through $D_7$–$D_0$) |
| 1 | 1 | No banks active |

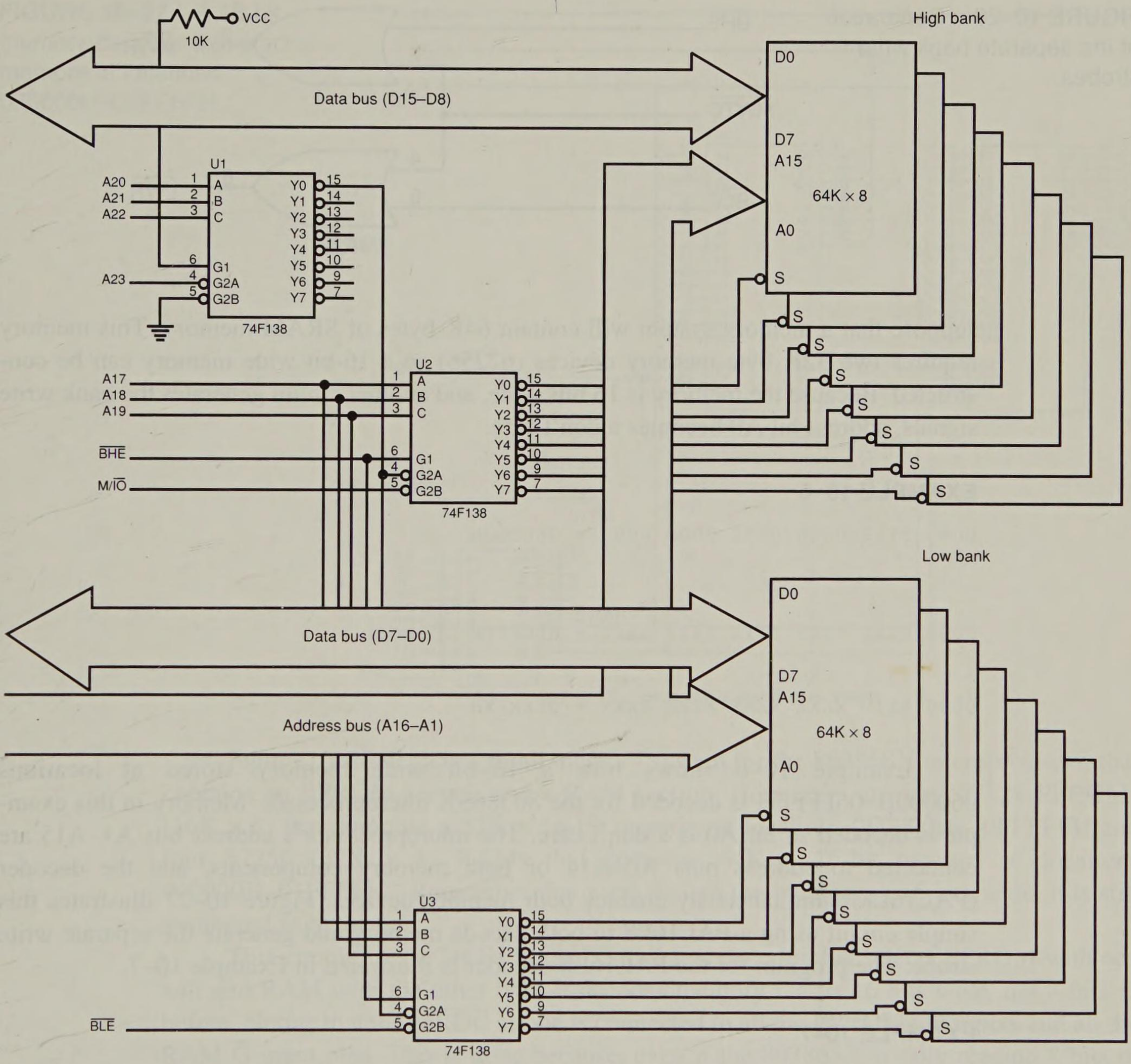



**FIGURE 10–25** A 1M × 8 memory system for the 80386SX microprocessor using separate bank decoders.

always presented to the data bus during a read, the microprocessor ignores the 8-bit section it doesn't need without any conflicts or special problems.

Figure 10–26 depicts the generation of separate 80386SX write strobes for the memory system. Here, a 74LS32 OR gate combines $\overline{\text{BLE}}$ with $\overline{\text{MWTC}}$ (memory write control) for the *low bank write* selection signal ($\overline{\text{LWR}}$), and combines $\overline{\text{BHE}}$ with $\overline{\text{MWTC}}$ for the *high bank write* selection signal ($\overline{\text{HWR}}$).

A memory system that uses separate write strobes is constructed differently from either the 8-bit system (8088) or the system using separate memory banks. Memory, in a system that uses separate write strobes, is decoded as 16-bit wide memory. For example,

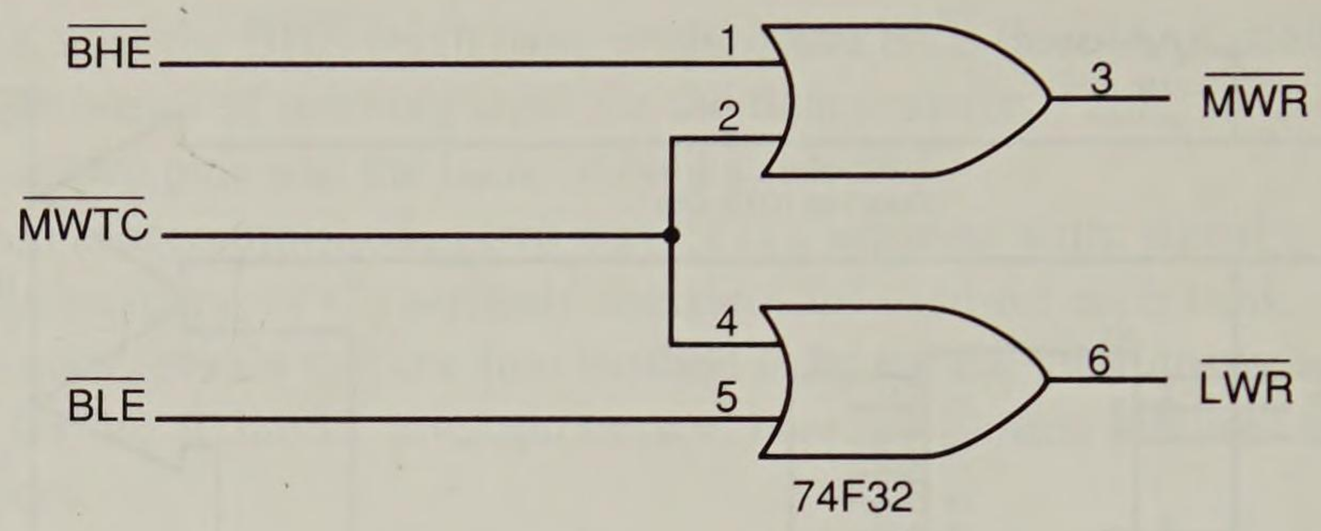



**FIGURE 10–26** Generation of the separate bank write strobes.

suppose that a memory system will contain 64K bytes of SRAM memory. This memory requires two 32K byte memory devices (62256) so a 16-bit wide memory can be constructed. Because the memory is 16 bits wide, and another circuit generates the bank write signals, address bit A0 becomes a don't care.

**EXAMPLE 10–6**

```
0000 0110 0000 0000 0000 0000 = 060000H

                to

0000 0110 1111 1111 1111 1111 = 06FFFFH


0000 0110 XXXX XXXX XXXX XXXX = 06XXXXH
```

Example 10–6 shows how a 16–bit wide memory stored at locations 060000H–06FFFFH is decoded for the 80386SX microprocessor. Memory in this example is decoded so bit A0 is a don't care. The microprocessor's address bits A1–A15 are connected to address pins A0–A14 of both memory components, and the decoder (PAL16L8) simultaneously enables both memory devices. Figure 10–27 illustrates this simple circuit using a PAL16L8 to both decode memory and generate the separate write strobe. The program for the PAL16L8 decoder is illustrated in Example 10–7.

**EXAMPLE 10–7**

```
TITLE       Address Decoder
PATTERN     Test 1R
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        6/24/93
CHIP        Decoder1R PAL16L8

;pins  1   2   3   4   5   6   7   8   9  10
       A23 A22 A21 A20 A19 A18 A17 A16 A0 GND

;pins   11  12  13  14  15  16  17   18 19  20
        BHE SEL LWR HWR NC  NC  MWTC NC NC  VCC

EQUATIONS

/SEL = /A23 * /A22 * /A21 * /A20 * /A19 * A18 * A17 * /A16
/LWR = /MWTC * /A0
/HWR = /MWTC * /BHE
```

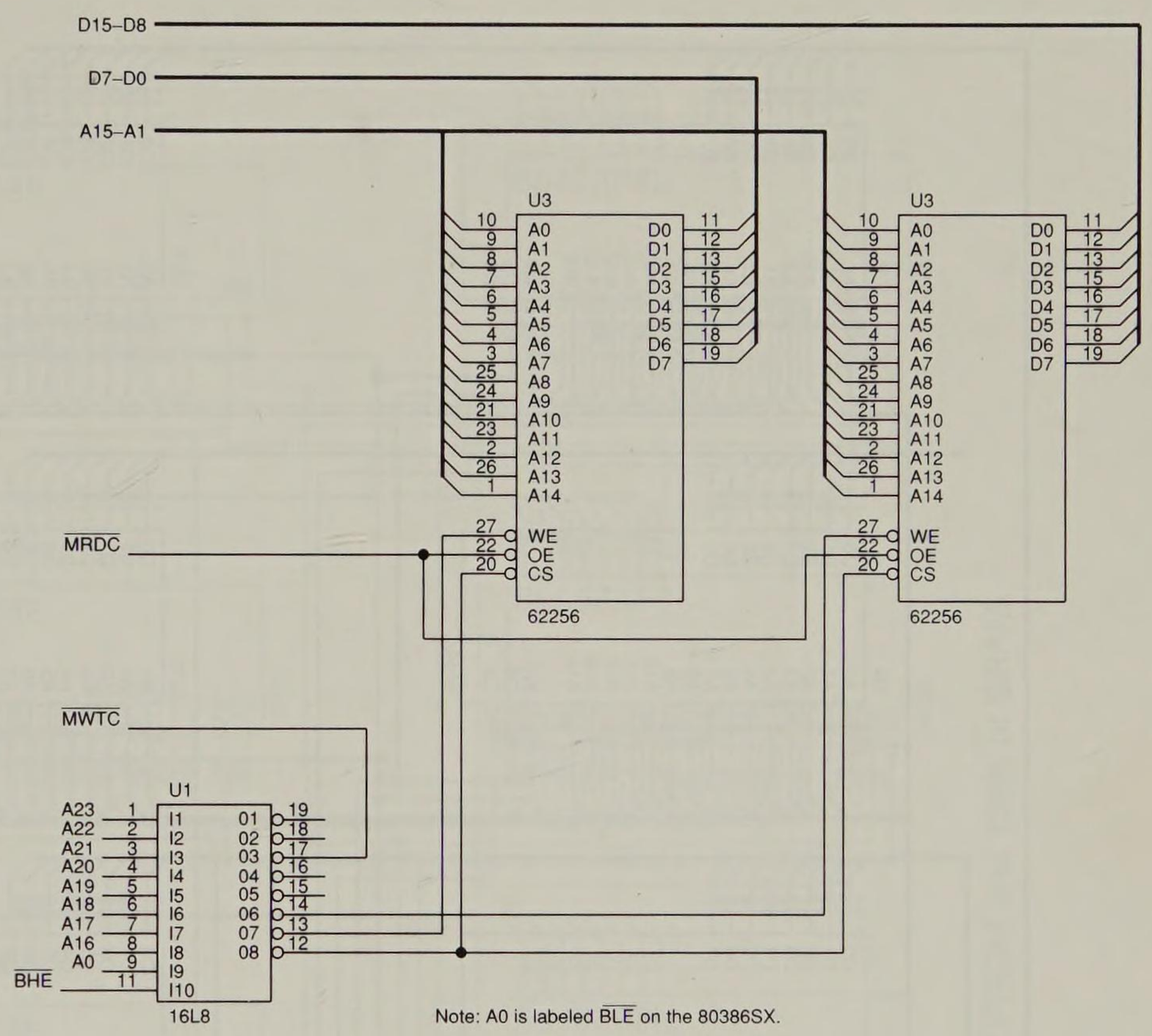



**FIGURE 10–27** A 16-bit memory decoder that places memory at locations 060000H–06FFFFH.

Figure 10–28 depicts a small memory system for the 80386SX microprocessor that contains an EPROM section and a RAM section. Here there are four 27128 EPROMs (16K × 8) that compose a 32K × 16-bit memory at locations 0F0000H—0FFFFFH and four 62256 (32K × 8) RAMs that compose 64K × 16-bit memory at locations 000000H–01FFFFH. (Remember that even though the memory is 16 bits wide, it is still numbered in bytes.)

This circuit uses a 74LS139 dual 2-to-4 line decoder that selects EPROM with one half and RAM with the other half. It decodes memory that is 16 bits wide, not 8 bits as before. Notice that the $\overline{\text{MRDC}}$ strobe is connected to all the EPROM $\overline{\text{OE}}$ inputs and all the RAM $\overline{\text{G}}$ input pins. This is done because, even if the 80386SX is only reading 8 bits of data, the application of the remaining 8 bits to the data bus has no effect on the operation of the 80386SX.

The $\overline{\text{LWR}}$ and $\overline{\text{HWR}}$ strobes are connected to different banks of the RAM memory. Here it does matter whether the microprocessor is doing a 16-bit or an 8-bit write. If the 80386SX writes a 16-bit number to memory, both $\overline{\text{LWR}}$ and $\overline{\text{HWR}}$ go low and enable the $\overline{\text{W}}$ pins in both upper and lower memory banks. But if the 80386SX does an 8-bit write, then only one of the write strobes goes low, writing to only one memory bank. The only time that the banks make a difference is for a memory write operation.

Notice that an EPROM decoder signal is sent to the 80386SX wait state generator because EPROM memory usually requires wait states. The signal comes from the NAND gate used to select the EPROM decoder section so that if EPROM is selected, a wait state is requested.

Figure 10–29 illustrates another memory system connected to the 80386SX microprocessor using a PAL16L8 as a decoder. This interface contains 256K bytes of EPROM

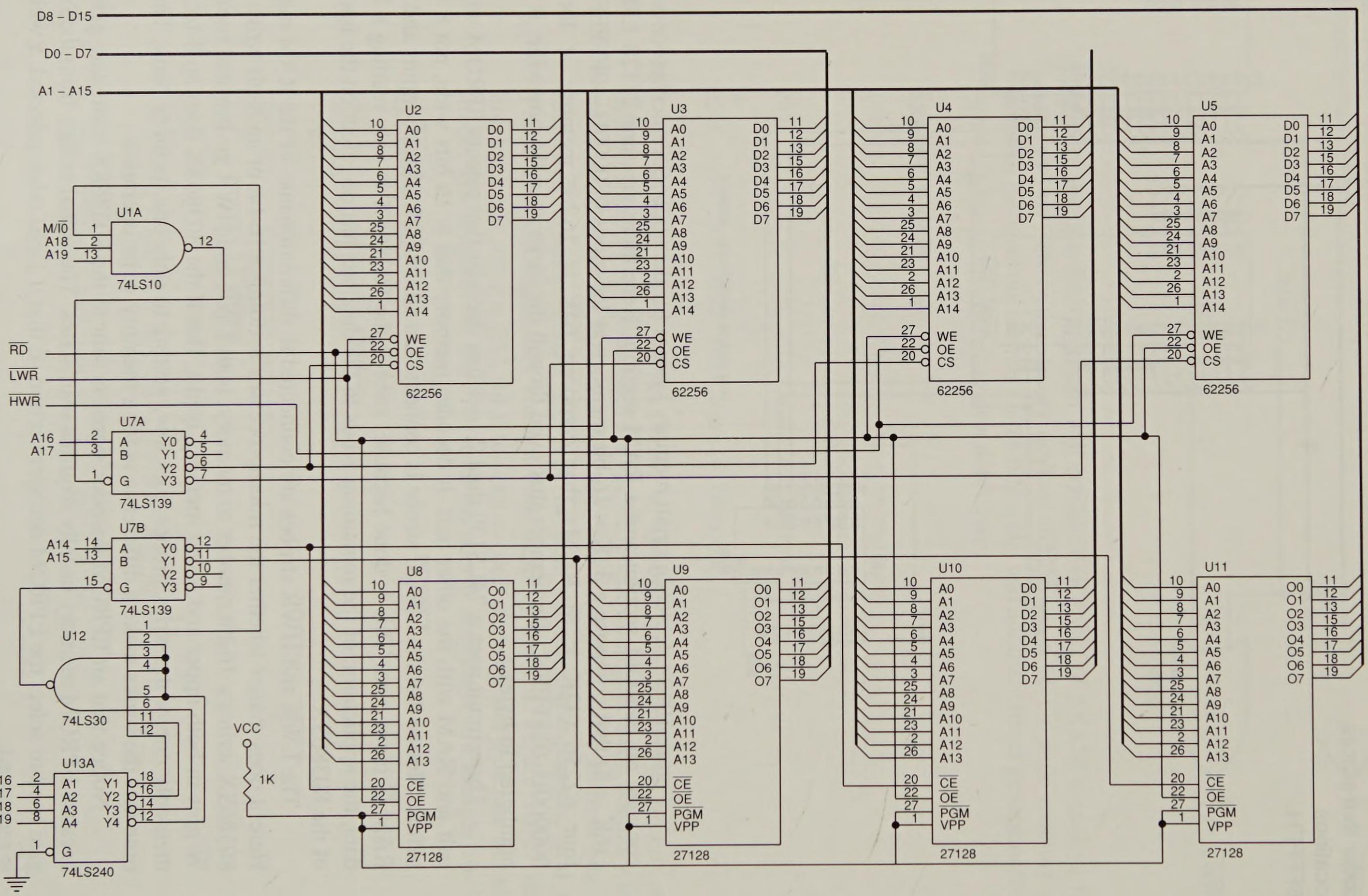



**FIGURE 10–28** A memory system for the 80386SX that contains a 64K-byte EPROM and a 128K-byte SRAM.

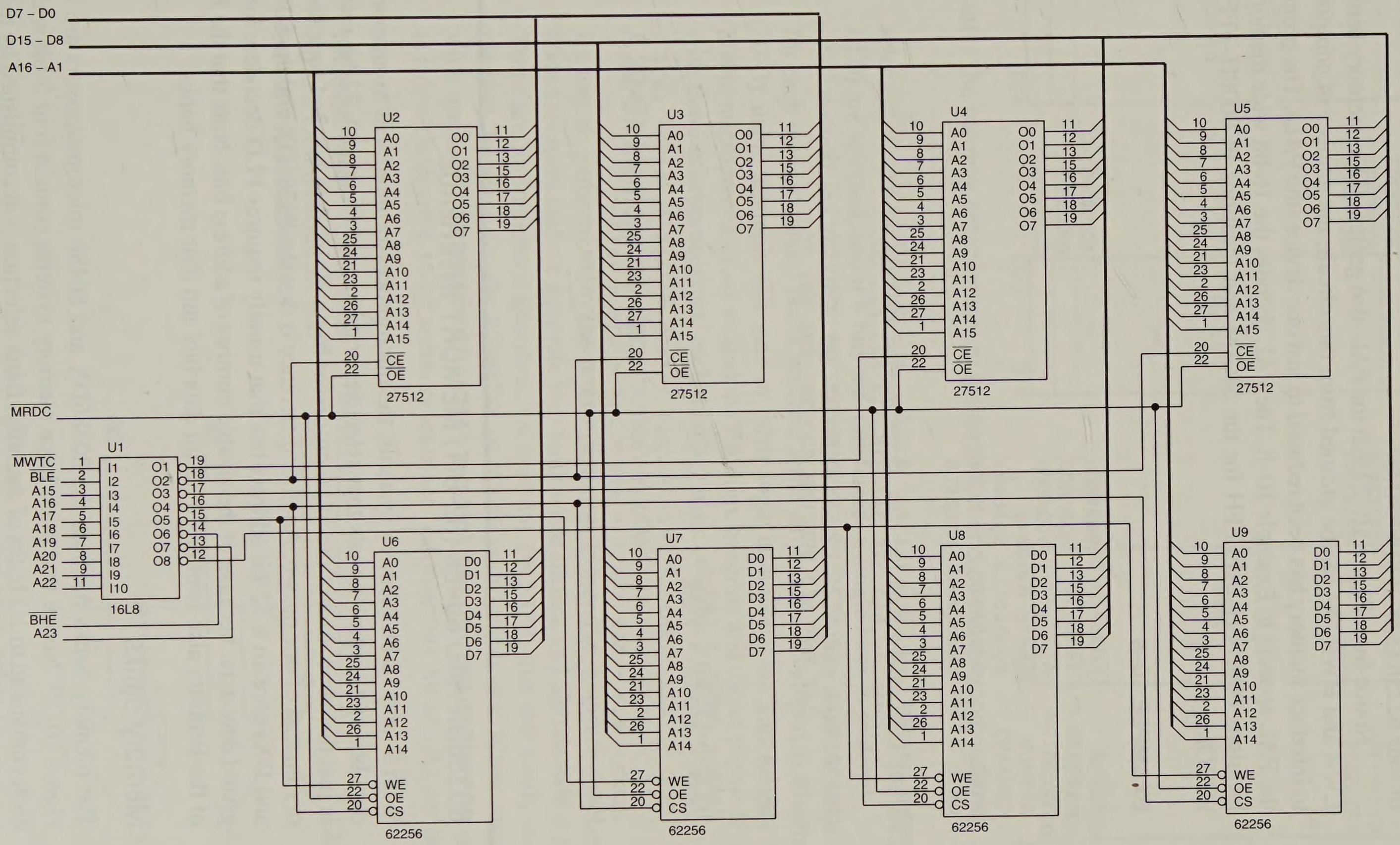



**FIGURE 10–29** An 80386SX memory system containing 256K of EPROM and 128K of SRAM.

in the form of four 27512 (64K × 8) EPROMs and 128K bytes of SRAM memory found in four 62256 (32K × 8) SRAMs.

Notice from Figure 10–29 that the PAL also generates the memory bank write signals $\overline{\text{LWR}}$ and $\overline{\text{HWR}}$. As can be gleaned from this circuit, the number of components required to interface memory has been reduced to just one device (the PAL). The program listing for the PAL appears in Example 10–8. The PAL decodes the 16-bit wide memory addresses at locations 000000H–01FFFFH for the SRAM and locations FC0000H–FFFFFFH for the EPROM.

**EXAMPLE 10–8**

```
TITLE       Address Decoder
PATTERN     Test 1Z
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        6/25/93
CHIP        Decoder1Z PAL16L8

;pins  1    2  3   4   5   6   7   8   9   10
       MWTC A0 A15 A16 A17 A18 A19 A20 A21 GND

;pins   11  12  13  14  15  16  17  18  19  20
        A22 HWR A23 BHE LWR RB0 RB1 EB0 EB1 VCC

EQUATIONS

/LWR = /MWTC * /A0
/HWR = /MWTC * /BHE
/RB0 = /A23 * /A22 * /A21 * /A20 * /A19 * /A18 * /A17 * /A16
/RB1 = /A23 * /A22 * /A21 * /A20 * /A19 * /A18 * /A17 * A16
/EB0 = A23 * A22 * A21 * A20 * A19 * A18 * /A17
/EB1 = A23 * A22 * A21 * A20 * A19 * A18 * A17
```

## 10–5 80386DX AND 80486 (32-BIT) MEMORY INTERFACE

With 8- and 16-bit memory systems, the microprocessor interfaces to memory through its data bus and control signals that select separate memory banks. A 32-bit memory system is just the same, except the microprocessor has a 32-bit data bus and four banks of memory instead of one or two. Another difference is that the 80386DX and the 80486 (both SX and DX) contain a 32-bit address bus that usually requires PLD decoders instead of integrated decoders, because of the sizable number of address bits. Note that the P24T version of the Pentium also contains a 32-bit data bus and four memory banks.

### Memory Banks

The memory banks for both the 80386DX and 80486 microprocessors are illustrated in Figure 10–30. Notice that these large memory systems contain four 8-bit wide banks that each contain up to 1G bytes of memory. Bank selection is accomplished by the bank selection signals, $\overline{\text{BE3}}$, $\overline{\text{BE2}}$, $\overline{\text{BE1}}$, and $\overline{\text{BE0}}$. If a 32-bit number is transferred, all four banks are

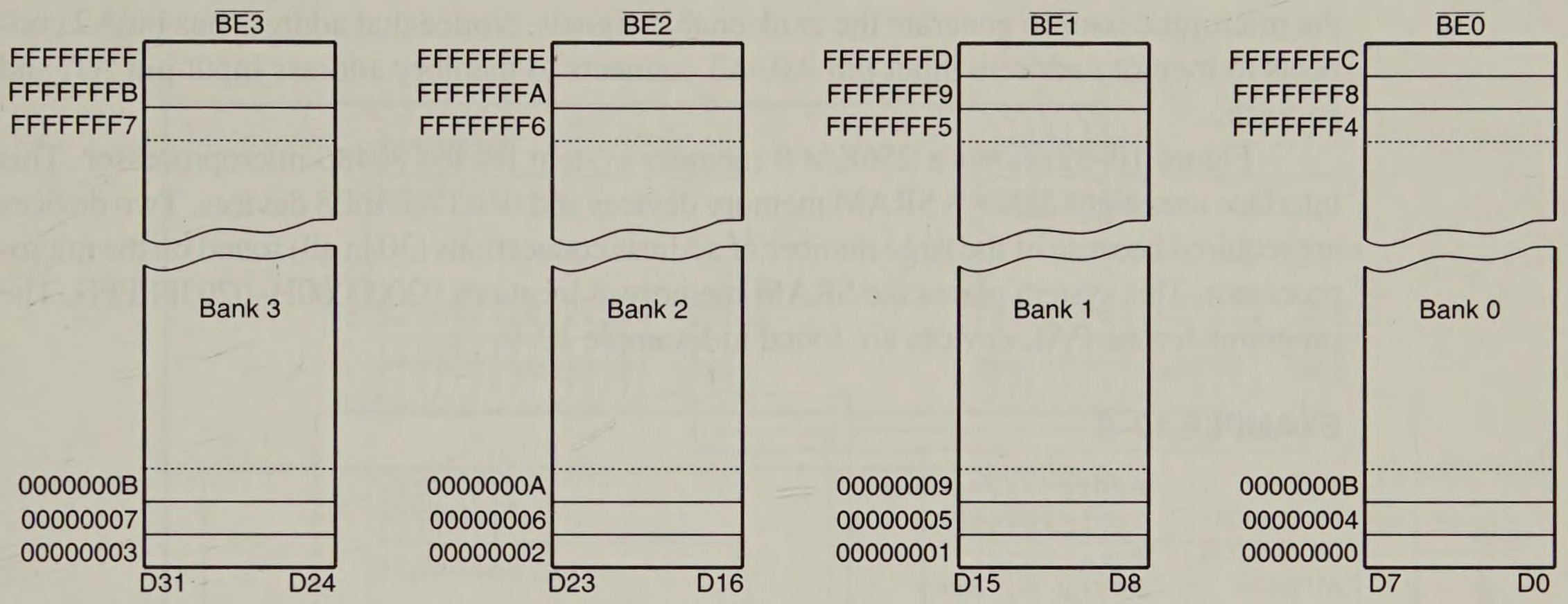



**FIGURE 10–30** The memory organization for the 80386DX and 80486 microprocessors.

selected; if a 16-bit number is transferred, two banks (usually $\overline{BE3}$ and $\overline{BE2}$ or $\overline{BE1}$ and BE0) are selected; and if 8 bits are transferred, a single bank is selected.

As with the 80386SX, the 80386DX and 80486 require separate write strobe signals for each memory bank. These separate write strobes are developed as illustrated in Figure 10–31 using a simple OR gate or other logic component such as a PAL. Note that the 80486 microprocessor, as well as the Pentium, generate and detect parity. If parity is in use with these microprocessors, each memory bank contains a ninth parity bit.

## A 32-Bit Memory Interface

As can be gathered from the prior discussion, a memory interface for the 80386DX or 80486 requires that we generate four bank write strobes and also decode a 32-bit address. There are no integrated decoders, such as the 74LS138, that can easily accommodate a memory interface for the 80386DX or 80486 microprocessors. When interfacing 32-bit wide memory, address bits A0 and A1 are don't cares for the memory decoder just as A0 is a don't care for a 16-bit memory interface. Address bits A0 and A1 are decoded within

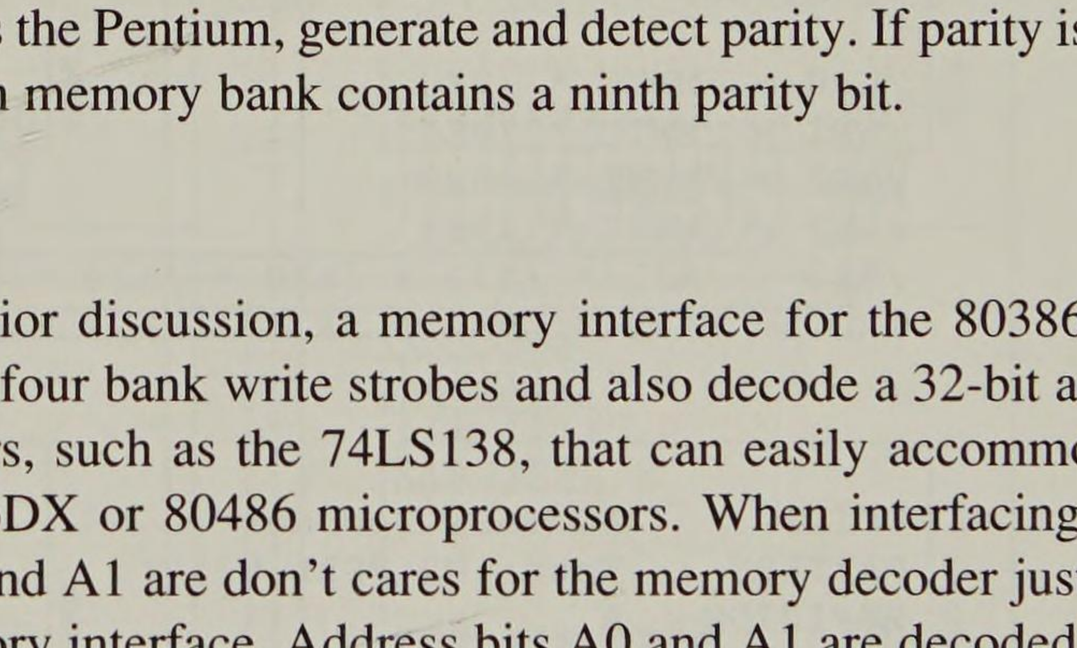



**FIGURE 10–31** Bank write signals for the 80386DX and 80486 microprocessors.

the microprocessor to generate the bank enable signals. Notice that address bus bit A2 connects to memory address input pin A0, A3 connects to memory address input pin A1, and so forth.

Figure 10–32 shows a 256K × 8 memory system for the 80486 microprocessor. This interface uses eight 32K × 8 SRAM memory devices and two PAL16L8 devices. Two devices are required because of the large number of address connections (30 in all) found on the microprocessor. This system places the SRAM memory at locations 02000000H–0203FFFFH. The programs for the PAL devices are found in Example 10–9.

**EXAMPLE 10–9**

```
TITLE       Address Decoder
PATTERN     Test 1U1 (PAL U1)
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        6/26/93
CHIP        Decoder1U1 PAL16L8

;pins  1    2   3   4   5   6   7   8   9   10
       MWTC BE0 BE1 BE2 BE3 A17 A28 A29 A30 GND

;pins   11  12  13 14 15  16  17  18  19  20
        A31 RB1 U2 NC WR0 WR1 WR2 WR3 RB0 VCC

EQUATIONS

/WR0 = /MWTC * /BE0
/WR1 = /MWTC * /BE1
/WR2 = /MWTC * /BE2
/WR3 = /MWTC * /BE3
/RB0 = /A31 * /A32 * /A30 * /A29 * /A28 * /A17 * /U2
/RB1 = /A31 * /A32 * /A30 * /A29 * /A28 * A17 * /U2


TITLE       Address Decoder
PATTERN     Test 1U2 (PAL U2)
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        6/26/93
CHIP        Decoder1U2 PAL16L8

;pins  1    2    3    4    5    6    7    8    9    10
       A18 A19 A20 A21 A22 A23 A24 A25 A26 GND

;pins   11  12 13 14 15 16 17 18 19 20
        A27 U2 NC NC NC NC NC NC NC VCC

EQUATIONS

/U2 = /A27 * /A26 * A25 * /A24 * /A23 * /A22 * /A21 * /A20 * /A19 * /A18
```

Although wait states are not mentioned in this section of the text, the 80386DX and 80486 microprocessors operate with very high clock rates that usually require wait states for memory access. Access time calculations for these microprocessors are discussed in

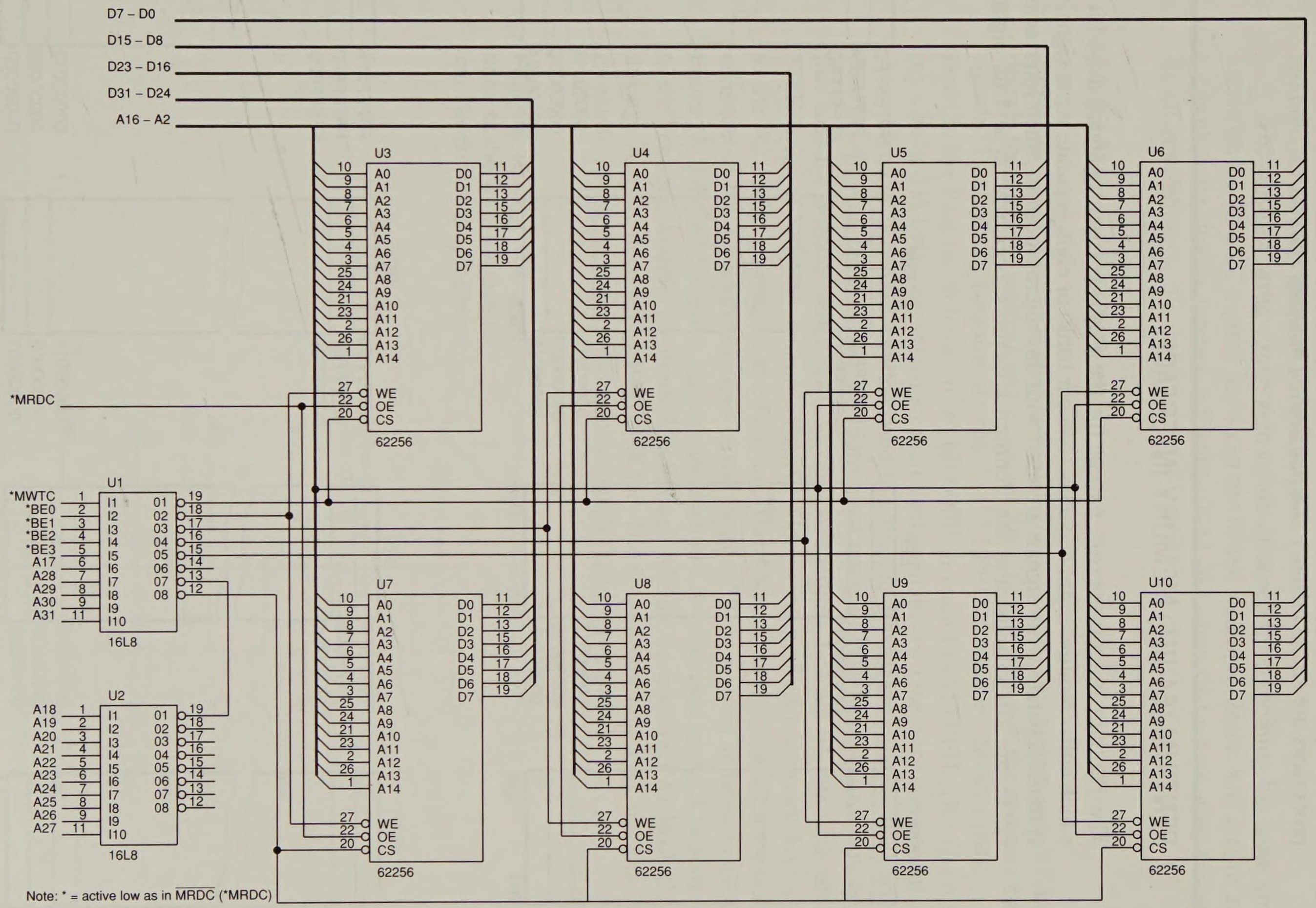



**FIGURE 10–32** A small 256K SRAM memory system interfaced to the 80486 microprocessor.

Chapter 9. The interface provides a signal used with the wait state generator that is not illustrated in this section of the text. Other devices found with these higher speed microprocessors are *cache memory* and *interleaved memory systems*.

## 10–6 PENTIUM (64-BIT) MEMORY INTERFACE

The Pentium microprocessor, except for the P24T version, contains a 64-bit data bus which either requires eight decoders (one per bank) or eight separate write signals. In most systems separate write signals are used with this microprocessor when interfacing memory. Figure 10–33 illustrates the Pentium's memory organization and its eight memory

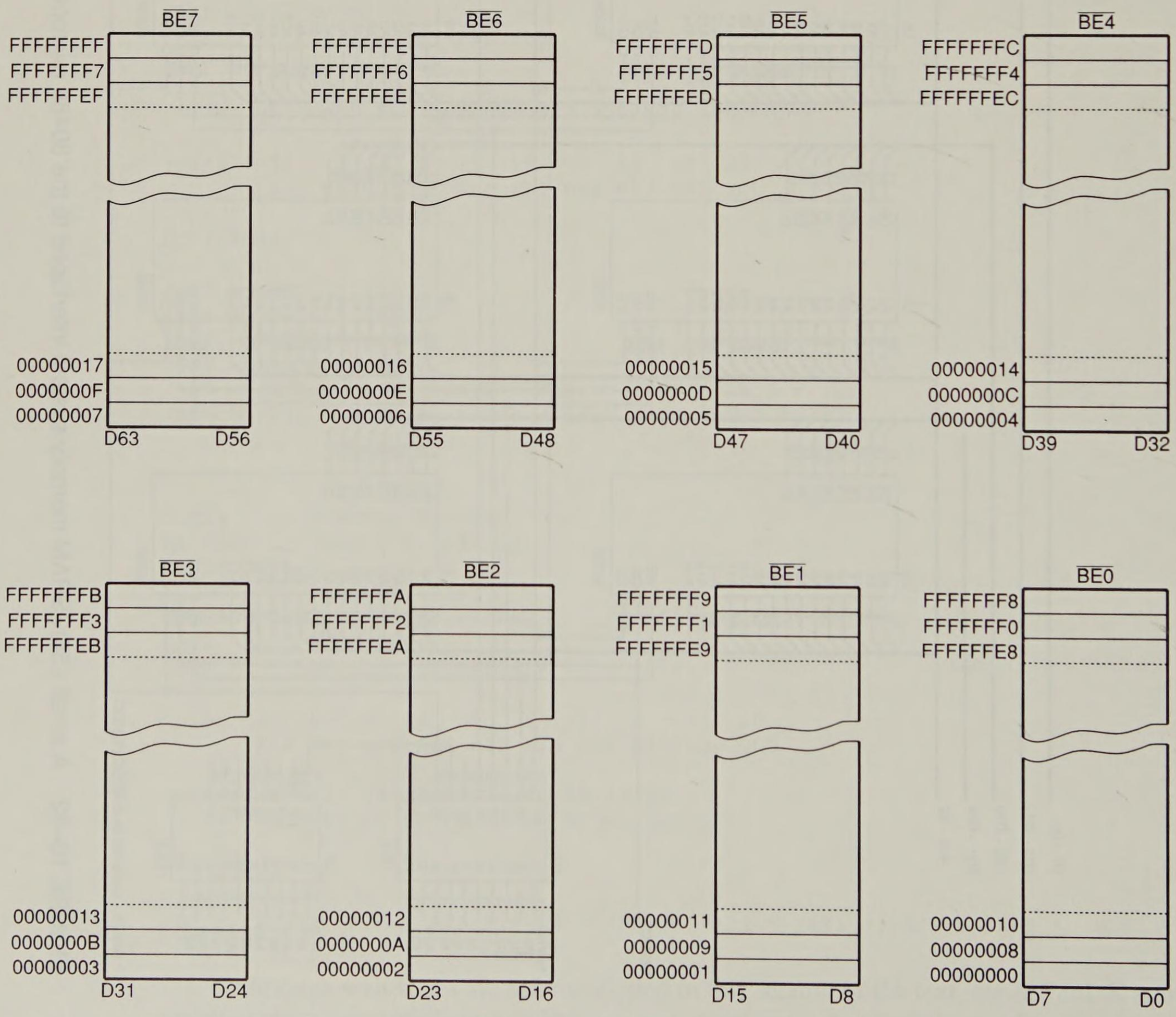



**FIGURE 10–33** The memory organization of the Pentium microprocessor.

banks. Notice that this is almost identical to the 80486 except it contains eight banks instead of four. As with earlier versions of the Intel microprocessor, this organization is required for upward memory compatibility.

The separate write strobe signals are obtained by combining the bank enable signals with the $\overline{\text{MWTC}}$ signal, just as in the 80486 microprocessor. The circuit employed for bank write signals appears in Figure 10–34. We often find a PAL used for bank write signal generation.

## 64-Bit Memory Interface

Figure 10–35 illustrates a small Pentium memory system. This system uses a PAL16L8 to decode the memory address and a second PAL16L8 to generate the separate bank write signals. This system contains sixteen 27512 EPROM memory devices (64K × 8) interfaced to the Pentium at locations FFF00000H through FFFFFFFFH. This is a total memory size of 1M bytes, organized so each bank contains two memory components.

Memory decoding, as illustrated in Example 10–10, is similar to the earlier examples except with the Pentium, the right-most three address bits (A2–A0) are ignored. In this case the decoder selects two sections of memory that are 64 bits wide and contain 512K bytes of EPROM memory. The A0 address input of each memory device connects to the A3 address output of the Pentium. The A1 address input of each memory device connects to the A4 address output of the Pentium. This skewed address connection continues until the A15 address input to the memory is connected to the A18 address output of the Pentium. Address positions A18 through A28 are decoded by PAL16L8 U2 and A29 through A31 are decoded by PAL16L8 U1. The program for both PAL devices is listed in Example 10–10 for memory locations F8000000H–FFFFFFFFH.

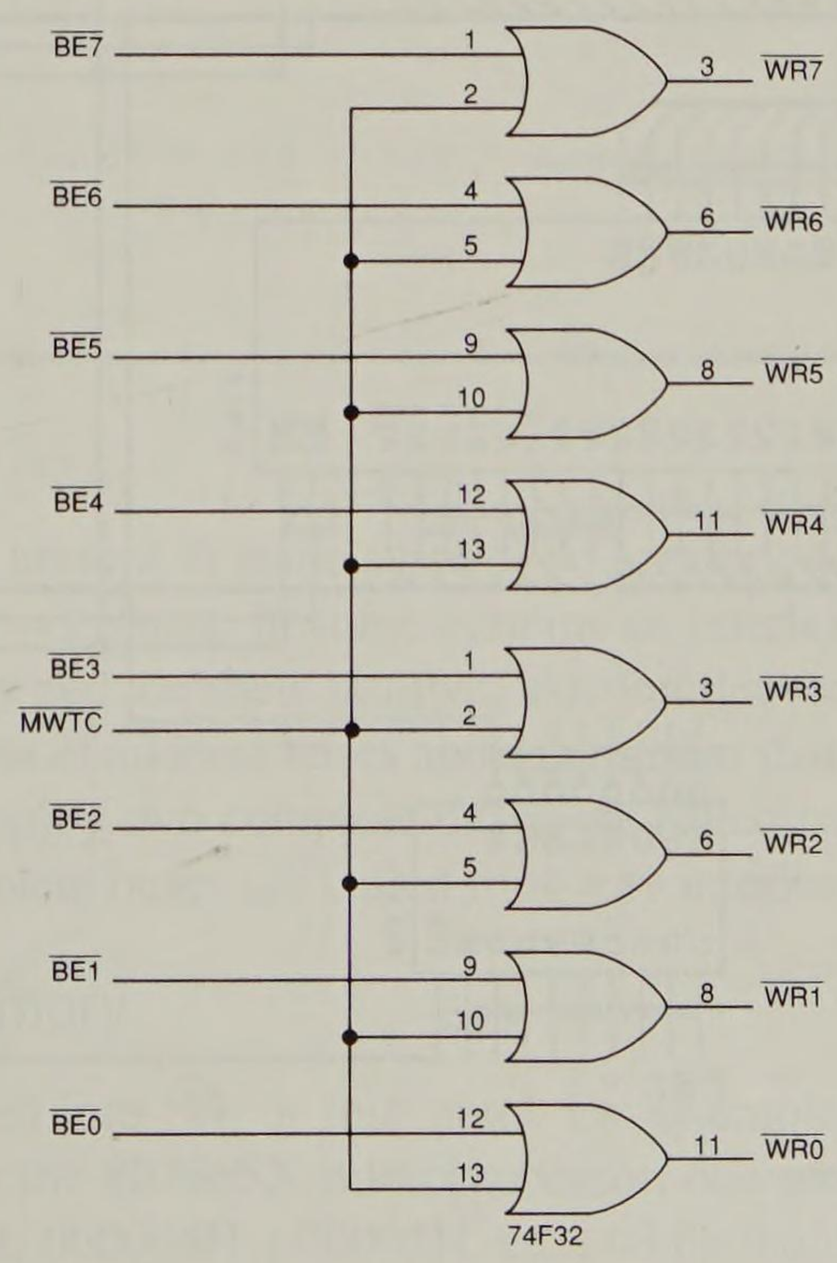



**FIGURE 10–34** The generation of the write strobes for the Pentium microprocessor.

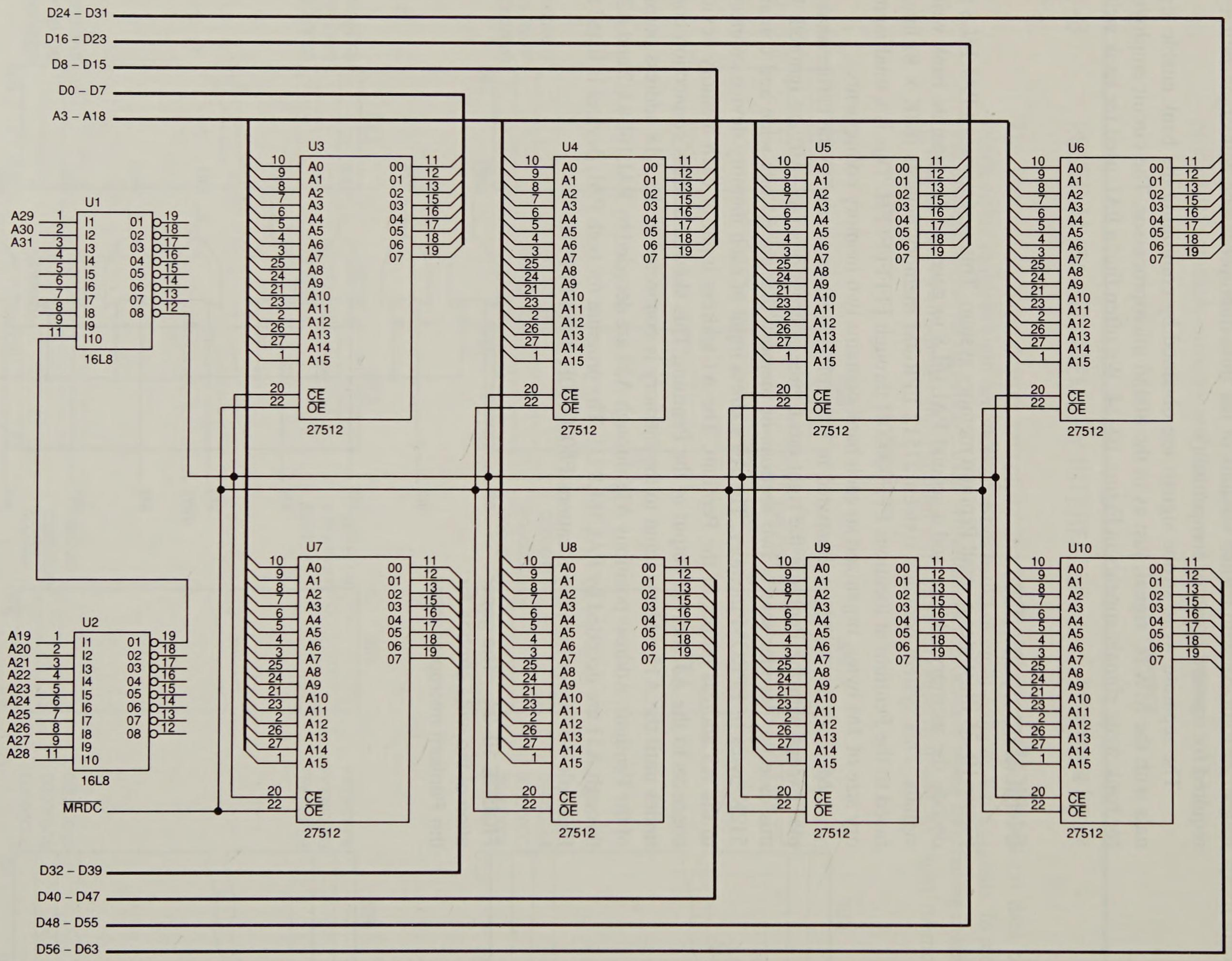



**FIGURE 10–35** A small 512K-byte EPROM memory interfaced to the Pentium microprocessor.

**EXAMPLE 10–10**

```
TITLE       Address Decoder
PATTERN     Test 1U4 (PAL U1)
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        6/27/93
CHIP        Decoder1U4 PAL16L8

;pins  1   2   3   4  5  6  7  8  9  10
       A29 A30 A31 NC NC NC NC NC NC GND

;pins   11 12 13 14 15 16 17 18 19  20
        U2 CE NC NC NC NC NC NC NC  VCC

EQUATIONS

/CE = /U2 * A29 * A30 * A31



TITLE       Address Decoder
PATTERN     Test 1U5 (PAL U2)
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        6/27/93
CHIP        Decoder1U5 PAL16L8

;pins  1   2   3   4   5   6   7   8   9   10
       A19 A20 A21 A22 A23 A24 A15 A26 A27 GND

;pins   11  12 13 14 15 16 17 18 19 20
        A28 NC NC NC NC NC NC NC U1 VCC

EQUATIONS

/U1 = A19 * A20 * A21 * A22 * A23 * A14 * A25 * A26 * A27 * A28
```

## 10–7 INTERLEAVED MEMORY

Interleaved memory systems are present in some systems so memory access times can be lengthened without the need for wait states. In some systems an interleaved memory may still require wait states, but may reduce their number. An interleaved memory system requires two or more complete sets of address buses and a controller that provides addresses for each bus. Systems that employ two complete buses are called two-way interleave, while systems that use four complete buses are called four-way interleave.

### Operation of Interleaved Memory

An interleaved memory is divided into two or four parts. For example, if an interleaved memory system is developed for the 80386SX microprocessor, one part contains the 16-bit addresses 000000H–000001H, 000004H–000005H, etc., while the other part contains

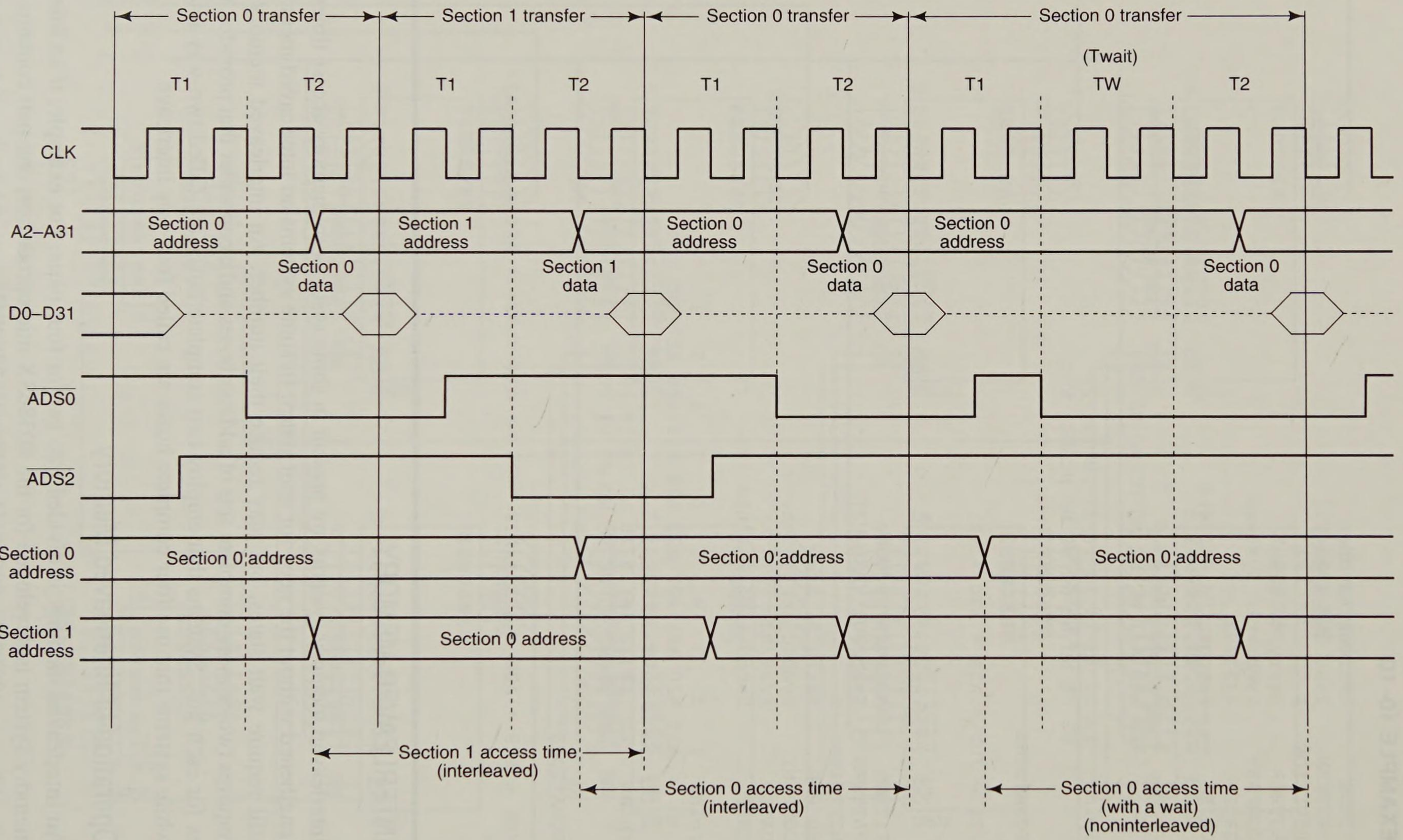



**FIGURE 10–36** The timing diagram of an interleaved memory system showing the access times and address signals for both sections of memory.

16-bit addresses 000002–000003, 000006H–000007H, etc. While the microprocessor accesses locations 000000H–000001H, the interleaved control logic generates the address strobe signal for locations 000002H–000003H. The interleaved control logic selects and accesses the word at locations 000002H–000003H, while the microprocessor processes the word at locations 000000H–000001H. This process alternates memory sections, increasing the performance of the memory system.

Interleaving lengthens the amount of access time provided to the memory, because the address is generated to select the memory before the microprocessor accesses it. This is because the microprocessor pipelines memory addresses, sending the next address out before the data are read from the last address.

The problem with interleaving is that the memory addresses must be accessed so each section is addressed alternately. This does not always happen as a program executes. Under normal program execution, the microprocessor alternately addresses memory approximately 93 percent of the time. The rest of the time, the microprocessor addresses data in the same memory section, which means that in 7 percent of the memory accesses, the memory system must cause wait states because of the reduced access time. The access time is reduced because the memory must wait until the previous data are transferred before it can obtain its address. This leaves it with less access time, therefore a wait state is required for accesses in the same memory bank.

Refer to Figure 10–36 for the timing diagram of the address as it appears at the microprocessor address pins. This timing diagram shows how the next address is output before the current data are accessed. It also shows how access time is increased using interleaved memory addresses for each section of memory compared to a noninterleaved access, which requires a wait state.

## The Interleaved Controller

Figure 10–37 pictures the interleaved controller. This is a fairly complex logic circuit that needs some explanation. First, if the SEL input (used to select this section of the memory) is inactive (logic 0), then the $\overline{\text{WAIT}}$ signal is a logic 1. Both ALE0 and ALE1, used to strobe the address to the memory sections, are logic 1s, causing the latches connected to them to become transparent.

As soon as the SEL input becomes a logic 1, this circuit begins to function. The A1 input is used to determine which latch (U2B or U5A) becomes a logic 0, selecting a section of the memory. Also the ALE pin, which becomes a logic 0, is compared with the previous state of the ALE pins. If the same section of memory is accessed a second time, the $\overline{\text{WAIT}}$ signal becomes a logic 0 requesting a wait state.

## An Interleaved Memory System

Figure 10–38 illustrates an interleaved memory system that uses the circuit of Figure 10–37. Notice how the ALE0 and ALE1 signals are used to capture the address for either section of memory. The memory in each bank is 16 bits wide. If accesses to memory require 8-bit data, then in most cases the system causes wait states. As a program executes, the 80386SX fetches instructions 16 bits at a time from normally sequential memory locations. Program execution uses interleaving in most cases. If a system is going to access mostly 8-bit data, it is doubtful that memory interleaving will reduce the number of wait states.

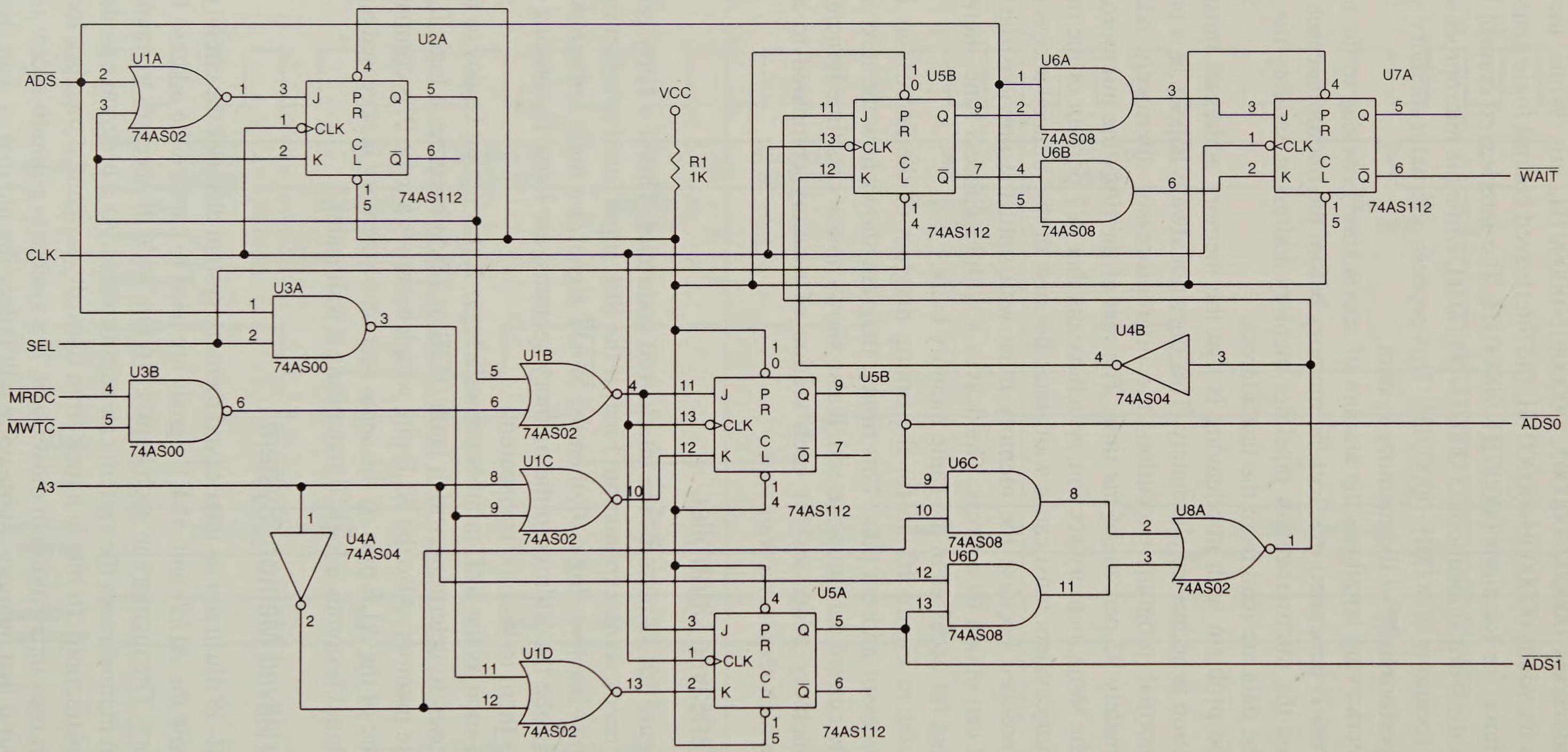



**FIGURE 10–37** The interleaved control logic, which generates separate $\overline{\text{ADS}}$ signals and a $\overline{\text{WAIT}}$ signal used to control interleaved memory.

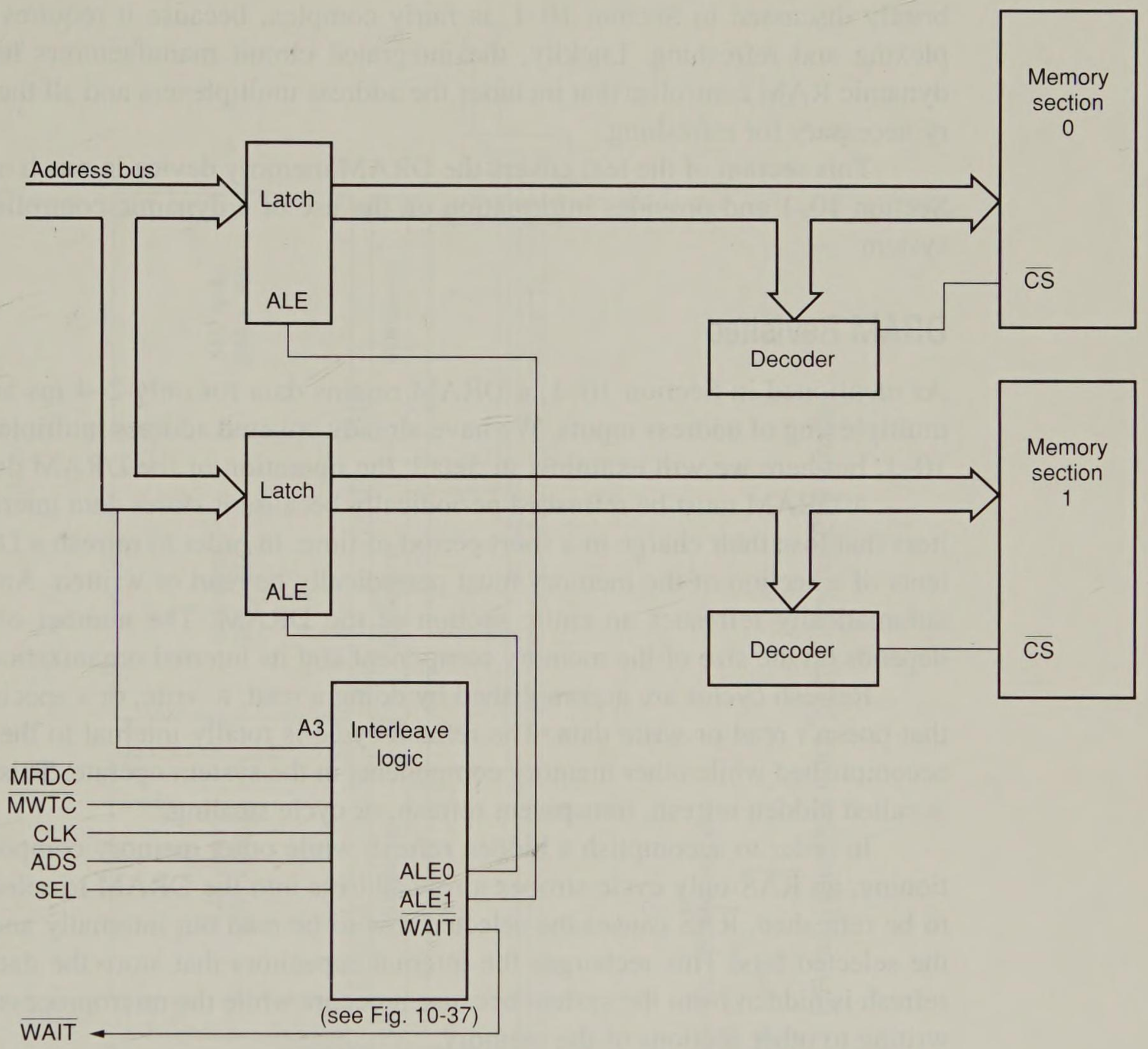



**FIGURE 10–38** An interleaved memory system showing the address latches and the interleaved logic circuit.

The access time allowed by an interleaved system such as the one shown in Figure 10–38 is increased to 112 ns from 69 ns, using a 16 MHz system clock. (If a wait state is inserted, access time with an 16 MHz clock is 136 ns, which means an interleaved system performs at about the same rate as a system with one wait state.) If the clock is increased to 20 MHz the interleaved memory requires 89.6 ns where standard, noninterleaved memory interfaces allow 48 ns for memory access. At this higher clock rate, 80 ns DRAMs function properly, without wait states when the memory addresses are interleaved. If an access to the same section occurs, then a wait state is inserted.

## 10–8 DYNAMIC RAM

Because RAM memory is often very large, it requires many SRAM devices at a great cost or just a few DRAMs (dynamic RAMs) at a much reduced cost. The DRAM memory, as

briefly discussed in Section 10–1, is fairly complex, because it requires address multiplexing and refreshing. Luckily, the integrated circuit manufacturers have provided a dynamic RAM controller that includes the address multiplexers and all the timing circuitry necessary for refreshing.

This section of the text covers the DRAM memory device in much more detail than Section 10–1 and provides information on the use of a dynamic controller in a memory system.

## DRAM Revisited

As mentioned in Section 10–1, a DRAM retains data for only 2–4 ms and requires the multiplexing of address inputs. We have already covered address multiplexers in Section 10–1, but here we will examine, in detail, the operation of the DRAM during refresh.

A DRAM must be refreshed periodically because it stores data internally on capacitors that lose their charge in a short period of time. In order to refresh a DRAM, the contents of a section of the memory must periodically be read or written. Any read or write automatically refreshes an entire section of the DRAM. The number of bits refreshed depends on the size of the memory component and its internal organization.

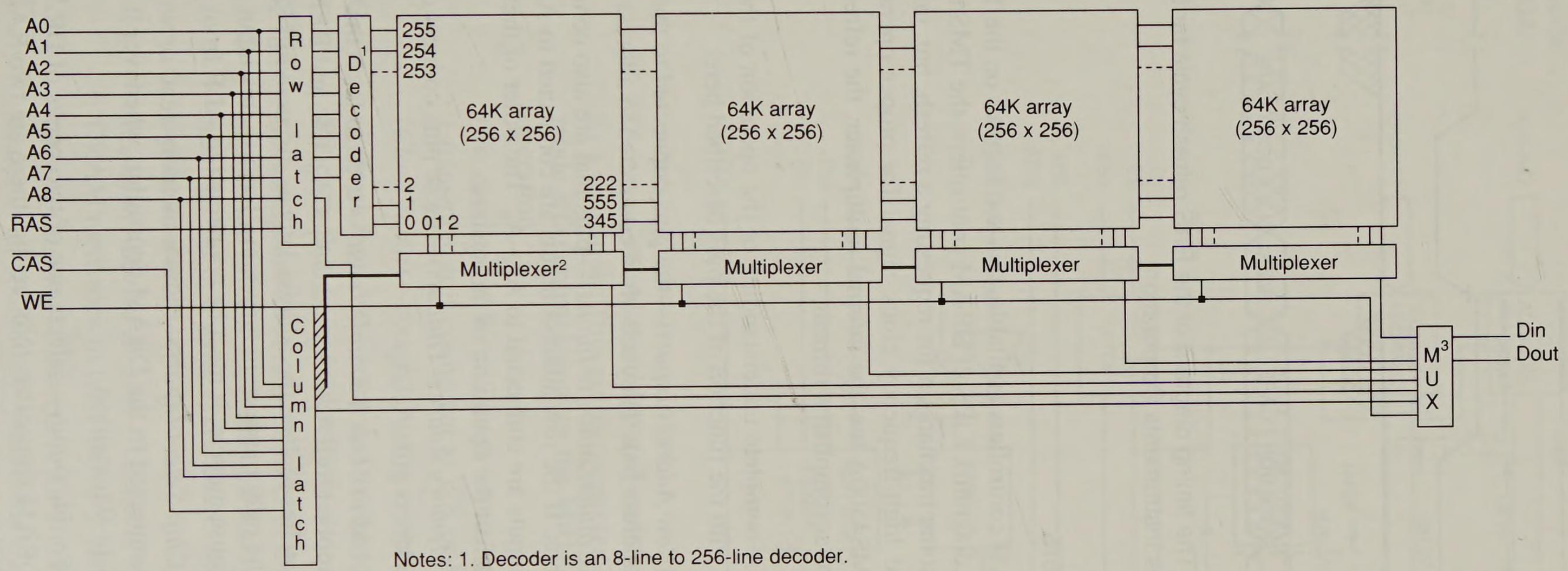
Refresh cycles are accomplished by doing a read, a write, or a special refresh cycle that doesn't read or write data. The refresh cycle is totally internal to the DRAM and is accomplished while other memory components in the system operate. This type of refresh is called hidden refresh, transparent refresh, or cycle stealing.

In order to accomplish a hidden refresh while other memory components are functioning, an $\overline{\text{RAS}}$-only cycle strobes a row address into the DRAM to select a row of bits to be refreshed. $\overline{\text{RAS}}$ causes the selected row to be read out internally and rewritten into the selected bits. This recharges the internal capacitors that store the data. This type of refresh is hidden from the system because it occurs while the microprocessor is reading or writing to other sections of the memory.

The DRAM's internal organization contains a series of rows and columns. A 256K $\times$ 1 DRAM has 256 columns, each containing 256 bits, or rows organized into four sections of 64K bits each. Whenever a memory location is addressed, the column address selects a column (or internal memory word) of 1,024 bits (one per section of the DRAM). Refer to Figure 10–39 for the internal structure of a 256K $\times$ 1 DRAM.

Figure 10–40 illustrates the timing for an $\overline{\text{RAS}}$-only refresh cycle. The main difference between the $\overline{\text{RAS}}$ and a read or write is that it applies only a refresh address, which is usually obtained from a 7- or 8-bit binary counter. The size of the counter is determined by the type of DRAM being refreshed. The refresh counter is incremented at the end of each refresh cycle so all the rows are refreshed in 2 or 4 ms, depending on the type of DRAM.

If there are 256 rows to be refreshed within 4 ms, as in a 256K $\times$ 1 DRAM, then the refresh cycle must be activated at least once every 15.6 µs in order to meet the refresh specification. For example, it takes the 80386SX, running at a 20 MHz clock rate, 100 ns to do a read or a write. Because the DRAM must have a refresh cycle every 7.8 µs, for a 4 ms refresh interval, this means that for every 13 or so memory reads or writes, the memory system must run a refresh cycle or memory data will be lost. This represents a loss of 7.8 percent of the computer's time, a small price to pay for the savings represented by using the dynamic RAM.




Notes: 1. Decoder is an 8-line to 256-line decoder.
2. Multiplexor is 256 to 1 line.
3. Multiplexor is 4 to 1 line.

**FIGURE 10–39** The internal structure of a 256K $\times$ 1 DRAM. Note that each of the internal 256 words are 1,025 bits wide.

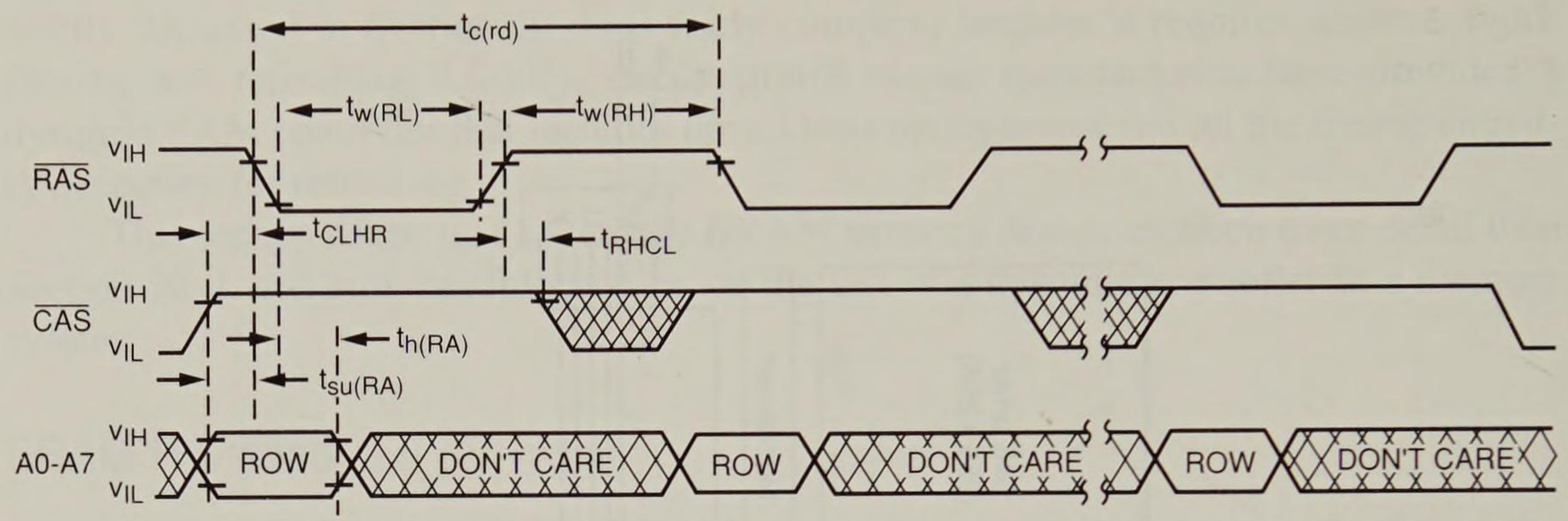



**FIGURE 10–40** The timing diagram of the $\overline{RAS}$ refresh cycle for the TMS4464 DRAM. (Courtesy of Texas Instruments Corporation)

## DRAM Controllers

Of the many DRAM controllers available, this text focuses on the TMS4500A (see Figure 10–41 for a block diagram). Like all DRAM controllers, the TMS4500A contains address multiplexers and some mechanism for requesting a refresh, but, unlike the others, it does not need a special high-frequency clock signal for proper operation during a refresh. Notice that the TMS4500A has the internal multiplexer, the refresh counter, and all the timing necessary to accomplish a refresh.

*Pin Description.* A complete understanding of the operation of the TMS4500A depends upon a familiarity with the function of the pins described here:

$RA_7$–$RA_0$ (*Row Address Inputs*)—the pins connected to the microprocessor's address bus. They are often connected to address bus bits $A_7$–$A_0$.

$CA_7$–$CA_0$ (*Column Address Inputs*)—pins that are also connected to the address bus. If the row address inputs are connected to $A_7$–$A_0$, then these inputs are connected to $A_{15}$–$A_8$. The order of these connections doesn't affect the operation of the system.

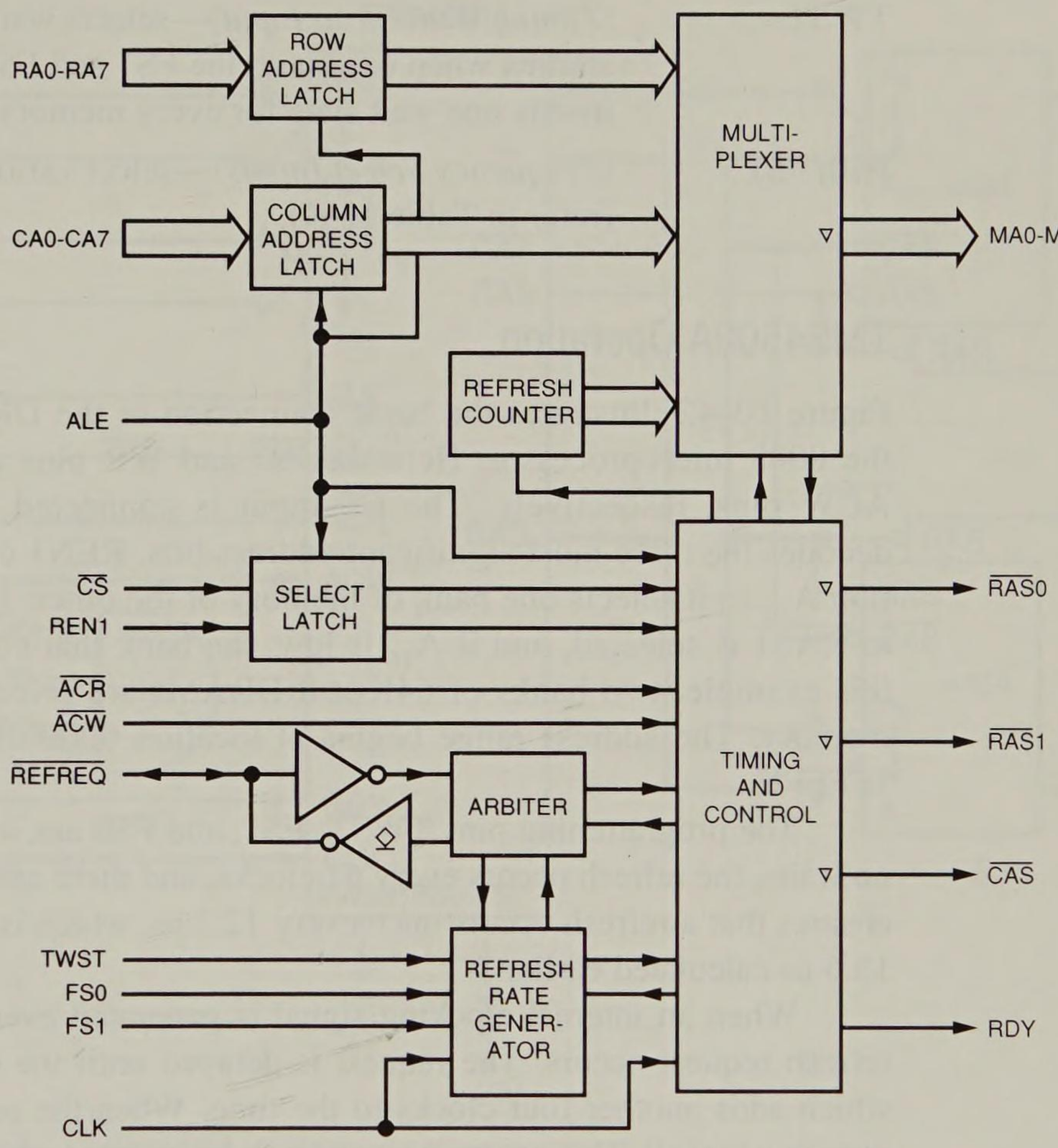
$MA_7$–$MA_0$ (*Memory Address Outputs*)—these pins connect directly to DRAM address pins $A_7$–$A_0$.

ALE (*Address Latch Enable Input*)—used to latch the 16 address inputs applied to $RA_7$–$RA_0$ and $CA_7$–$CA_0$, $\overline{CS}$, and REN1. Notice how these latches replace the address latches discussed in Chapter 7 if DRAM is the only memory in a system. Because this input must be active to cause a cycle, it must be connected to ALE from the microprocessor.

$\overline{CS}$ (*Chip Select Input*)—starts a memory read or write for the DRAMs connected to the DRAM controller, whenever it is a logic 0 at ALEs 1-to-0 transition.

REN1 (*Enable Input*)—selects one of two banks of DRAMs connected to the DRAM controller. (Notice that there are two $\overline{RAS}$ outputs.) When REN1 is high $\overline{RAS1}$ is selected, and when it is low $\overline{RAS2}$ is selected.

**FIGURE 10–41** The block diagram of the TMS4500A dynamic RAM controller. (Courtesy of Texas Instruments Corporation)




| | |
|---|---|
| $\overline{\text{ACR}}$ | (*Access Control Read Input*)—ends a memory read cycle on the 0-to-1 transition. This pin is connected to the 8086/8088 minimum mode $\overline{\text{RD}}$ control signal or the $\overline{\text{MRDC}}$ signal for maximum mode operation. |
| $\overline{\text{ACW}}$ | (*Access Control Write Input*)—ends a memory write cycle on the 0-to-1 transition. It is connected to $\overline{\text{WR}}$ or $\overline{\text{MWTC}}$. |
| CLK | (*Clock Input*)—connects to the system clock of the microprocessor. |
| REFREQ | (*Refresh Request*)—starts a refresh cycle or, when used as an output, indicates that an internal refresh cycle is in progress. |
| $\overline{\text{RAS1}}$, $\overline{\text{RAS0}}$ | (*Row Address Strobes*)—connect to the DRAM $\overline{\text{RAS}}$ inputs. Both signals are low for a refresh operation, but only one goes low during a normal read or a write cycle. REN1 selects which $\overline{\text{RAS}}$ pin goes low during a memory operation. |
| $\overline{\text{CAS}}$ | (*Column Address Strobe*)—connects to the $\overline{\text{CAS}}$ inputs of all of the DRAM memories in both memory banks. |
| RDY | (*Ready Output*)—connects to the RDY input of the 8284A clock generator in an 8086/8088 system. RDY becomes active when the DRAM controller performs an internal refresh cycle. |

TWST (*Timing/Wait Strap Input*)—selects waits and/or certain timing constraints when used with the FS1 and FS0 inputs. A logic 1 on this pin inserts one wait state for every memory access.

FS0, FS1 (*Frequency Select Inputs*)—select various mode and frequency options (refer to Table 10–4).

## TMS4500A Operation

Figure 10–42 illustrates the basic connection of the DRAM controller to the buses of the 8088 microprocessor. Here the $\overline{RD}$ and $\overline{WR}$ pins are connected to the $\overline{ACR}$ and $\overline{ACW}$ pins, respectively. The $\overline{CS}$ input is connected to a 4-input NAND gate that decodes the three most significant address bits. REN1 connects to address bus connection $A_{16}$ so it selects one bank of memory or the other. If $A_{16}$ is high, the bank attached to $\overline{RAS1}$ is selected, and if $A_{16}$ is low, the bank that connects to $\overline{RAS0}$ is selected. In this example, two banks of 64K × 8 DRAMs are selected for a total of 128K × 8 of memory. The address range begins at location 00000H and extends through location 1FFFFH.

The programming pins TWST, FS1, and FS0 are selected (0, 1, and 1) so there are no waits, the refresh occurs every 61 clocks, and there are 4 clocks for every refresh. This ensures that a refresh occurs once every 12.2 µs, which is well within the tolerance of the 15.6 µs calculated earlier.

When an internal clocking signal is generated every 61 clock periods, an internal refresh request occurs. The request is delayed until the current bus cycle is completed, which adds another four clocks to the time. When the request is honored, the RDY pin goes to a logic 0. This causes wait states to be inserted while the DRAM controller refreshes the memory. After the refresh is completed, the bus is free to do a read or a write until the next refresh request.

Figure 10–43 shows the timing diagram for the TMS4500A during the request for and completion of a refresh. Notice that the RDY line goes low to request wait states during the refresh operation.

**TABLE 10–4** Mode selection for the TMS4500 DRAM controller

| *TWST* | *FS1* | *FS0* | *Wait States* | *Refresh Rate* | *Minimum Clock (MHz)* | *Refresh Freq. (KHz)* | *Clocks per Refresh* |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | External | — | REFREQ | 4 |
| 0 | 0 | 1 | 0 | CLK ÷ 31 | 1.984 | 64–95 | 3 |
| 0 | 1 | 0 | 0 | CLK ÷ 46 | 2.944 | 64–85 | 3 |
| 0 | 1 | 1 | 0 | CLK ÷ 61 | 3.904 | 64–82 | 4 |
| 1 | 0 | 0 | 1 | CLK ÷ 46 | 2.944 | 64–85 | 3 |
| 1 | 0 | 1 | 1 | CLK ÷ 61 | 3.904 | 64–80 | 4 |
| 1 | 1 | 0 | 1 | CLK ÷ 76 | 4.864 | 64–77 | 4 |
| 1 | 1 | 1 | 1 | CLK ÷ 91 | 5.824 | 64–88 | 4 |

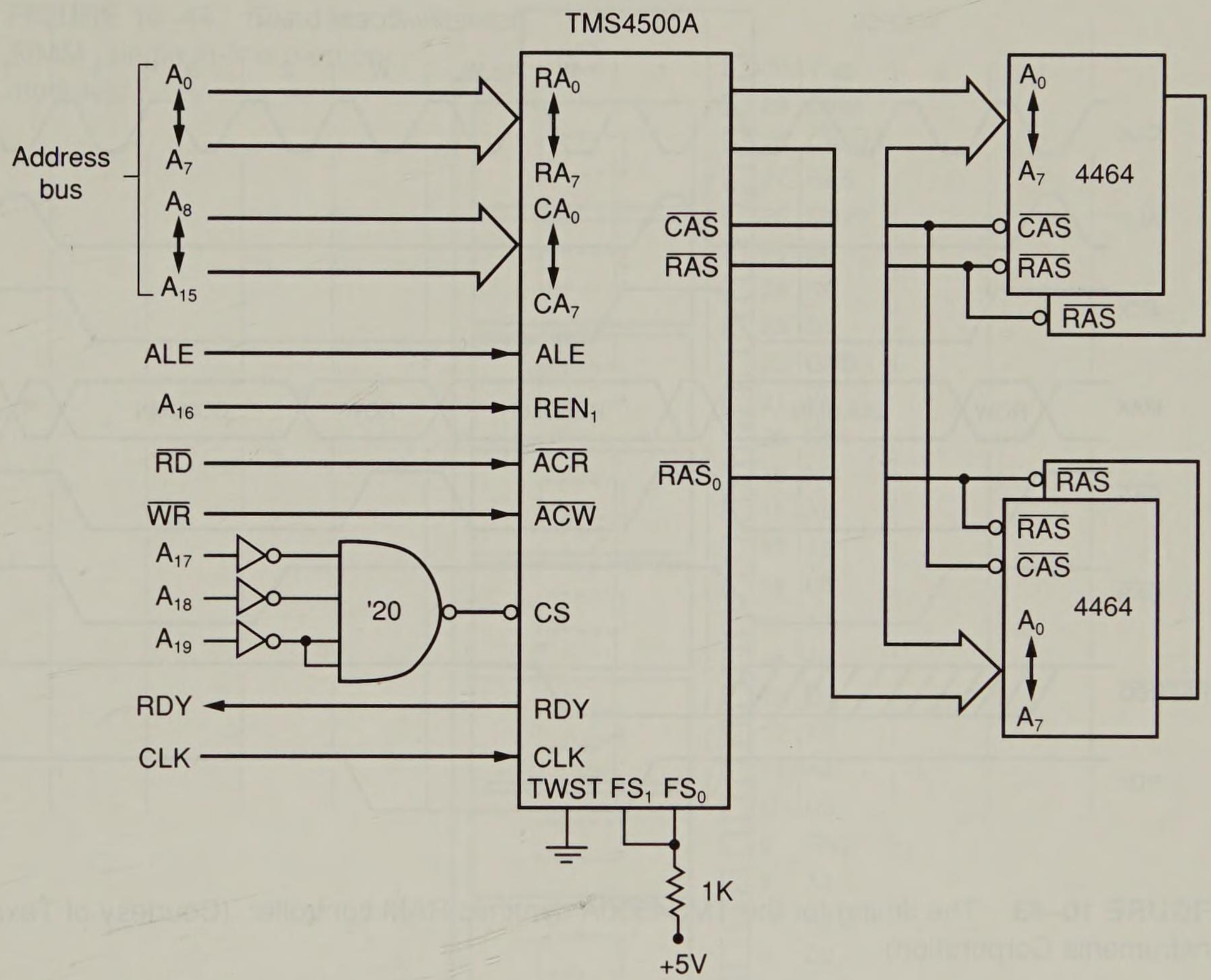



**FIGURE 10–42** The TMS4500A DRAM controller used to interface to 128K bytes of DRAM. Here, four TMS4464 DRAMs provide the memory located at locations 00000H–1FFFFH.

## DRAM Interface

Figure 10–44 illustrates a 1M × 9 SIMM* (*single in-line memory module*) that contains 10 address pins which are provided with a 20-bit multiplexed address. The $\overline{CAS}$ and $\overline{RAS}$ inputs are used to select ($\overline{CAS}$) and strobe addresses into the memory module. The $\overline{CAS}_9$ input selects the ninth parity bit. This device is also available as a 1M × 8 memory module, the 421000C8. The pins for the ninth parity bit on the 42100C9 are not connected on the 421000C8, and only 8 memory devices are present.

Most newer microprocessor-based computer systems use SIMM memory. Common sizes are the 256K × 9, 1M × 9, and 4M × 9. Soon the 16M × 9 SIMM will also become commonplace. Newer DRAM SIMM devices contain 72 pins and store information in a 32-bit wide format. For example, a 1M × 36 SIMM (4M bytes) contains four banks of memory, each 9 bits wide. These newer 72-pin SIMMs are ideal for use with the 80486 and Pentium microprocessors. The 72-pin SIMMs are available as 1M × 36, 2M × 36, 4M × 36, 8M × 36, and 16M × 36 formats. A variant of the SIMM, the SIP, is also available.

*SIMM (single in-line memory module) is a trade name of Wang Laboratories.

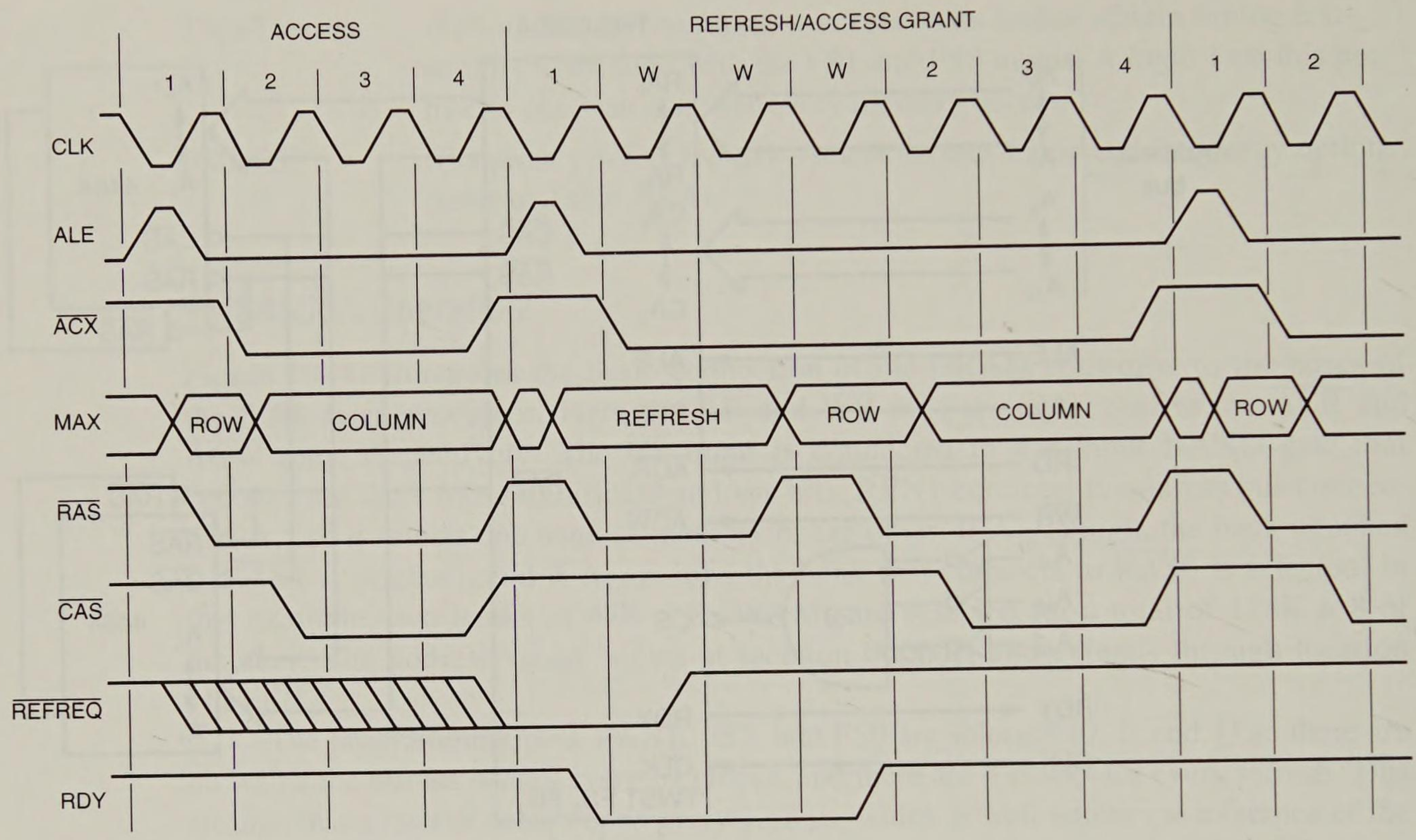


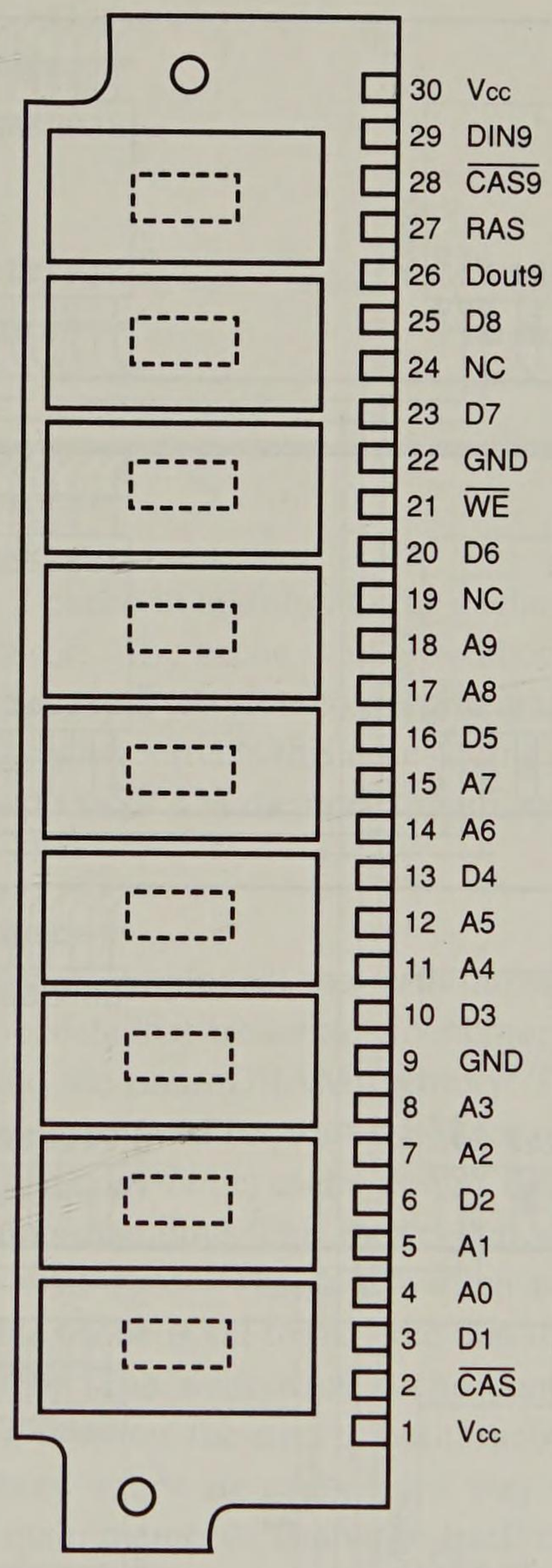

**FIGURE 10–43** The timing for the TMS4500A dynamic RAM controller. (Courtesy of Texas Instruments Corporation)

Figure 10–45 illustrates an 8M × 8 memory system designed for either the 80386DX or 80486 microprocessor. This system does not illustrate the the circuit that generates the $\overline{CAS}$ and $\overline{RAS}$ signals, but does show the address multiplexer. Notice again that the PAL16L8s are used (see Example 10–11) to develop the bank write strobe signals, and also to decode the memory address. Here the memory system is decoded to provide an 8M byte memory from location 00000000H through 007FFFFFH

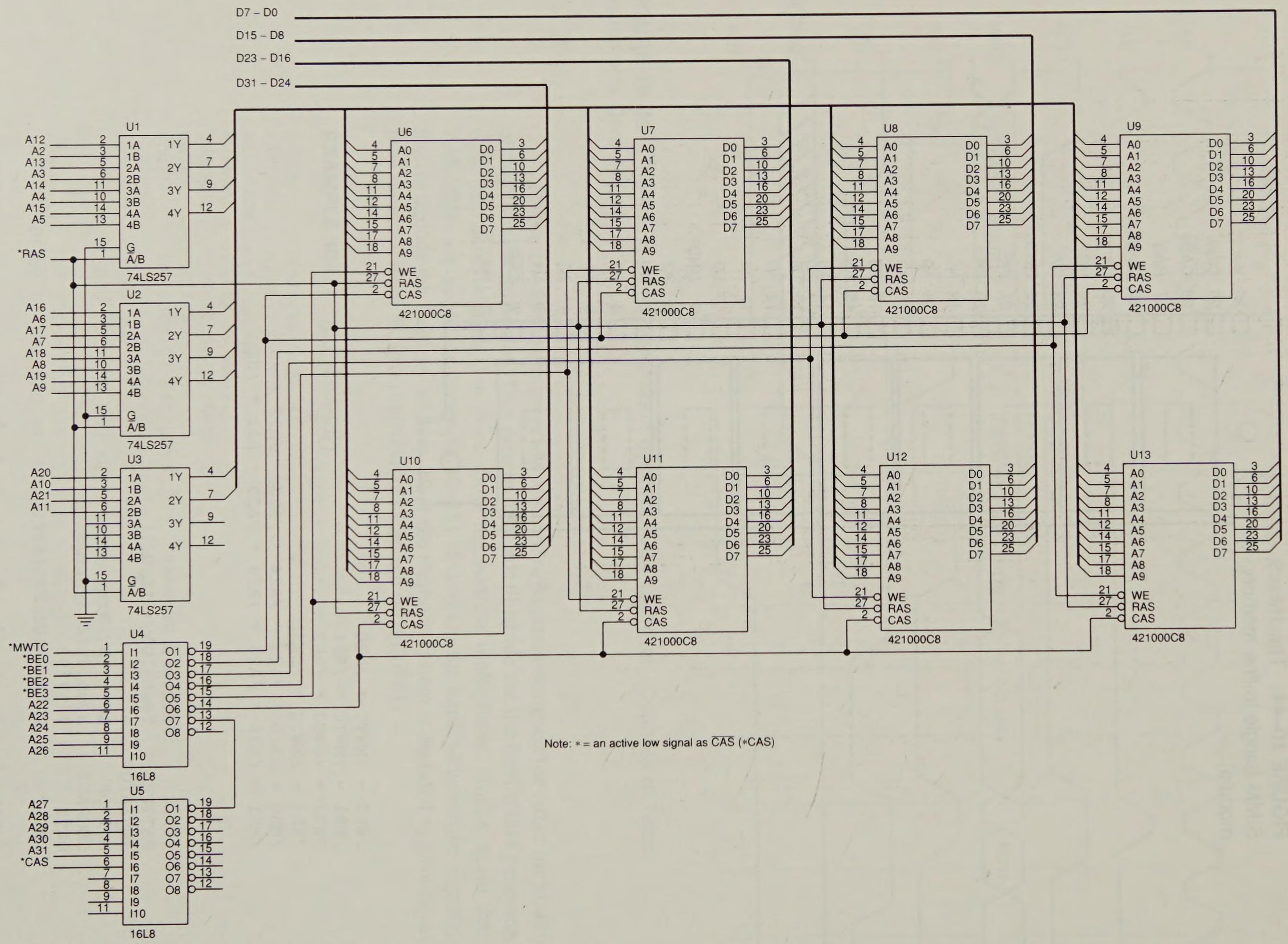
**EXAMPLE 10–11** (pp. 434–437)

```
TITLE       Address Decoder
PATTERN     Test 1U5 (PAL U1)
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        7/27/93
CHIP        Decoder1U4 PAL16L8

;pins  1    2   3   4   5   6   7   8   9   10
       MWTC BE0 BE1 BE2 BE3 A22 A23 A24 A25 GND

;pins  11  12 13 14  15  16  17  18  19  20
       A26 NC U2 MB1 WR0 WR1 WR2 WR3 MB0 VCC

EQUATIONS
```

**FIGURE 10–44** The 1M × 9 SIMM (single in-line memory module).




```
/WR0 = /MWTC * /BE0
/WR1 = /MWTC * /BE1
/WR2 = /MWTC * /BE2
/WR3 = /MWTC * /BE3
/MB0 = /A26 * /A25 * /A24 * /A23 * /A22 * /U2
/MB1 = /A26 * /A25 * /A24 * /A23 * A22 * /U2


TITLE       Address Decoder
PATTERN     Test 1U5 (PAL U2)
REVISION    A
AUTHOR      Barry B. Brey
COMPANY     Symbiotic Systems
DATE        7/27/93
CHIP        Decoder1U5 PAL16L8

;pins  1    2    3    4    5    6    7  8  9  10
```




Note: * = an active low signal as $\overline{CAS}$ (*CAS)

**FIGURE 10–45** An 8M DRAM memory system interfaced to the 80486 microprocessor.

```
        A27 A28 A29 A30 A31 CAS NC NC NC GND

;pins    11 12 13 14 15 16 17 18 19 20
         NC NC NC NC NC NC NC NC U2 VCC

EQUATIONS

/U2 = /A31 * /A30 * /A29 * /A28 * /A27 * /CAS
```

## 10–9 CACHE MEMORY

Most systems today use a level 2 cache in addition to the cache (level 1) found inside the 80486 and Pentium microprocessors. The cache is required because the memory system, constructed with DRAM components, is too slow to operate at the required access time of the microprocessor. The level 2 cache corrects this access time problem by storing often-used memory transfers in high-speed SRAM memory organized as a cache.

### Types of Caches

There are two main types of caches employed: the *write-through cache* and *write-back cache*. In the write-through cache, data that are written to memory are simultaneously written to both the SRAM cache and the main DRAM memory. This causes memory write operations to be slower because of the need for wait states for every write operation.

In the write-back (also called *copy-back*) cache design, data are written to the cache, but not to the main memory at the same time. This means that writing to memory appears to the microprocessor with no wait states. The data written to the write-back cache is stored so a bit associated with the cache is set to indicate that the information has not yet been written to the main memory. The write-back cache controller monitors the main memory, and when time allows, it writes the data to main memory without causing wait states to the microprocessor. Many writes are cached this way without the need to cause the microprocessor to wait for main memory. The write-back cache does not completely eliminate wait states, but greatly reduces their number.

Figure 10–46 shows the block diagram of a system that employs a cache memory. Notice that the cache is placed between the microprocessor and the main memory system.

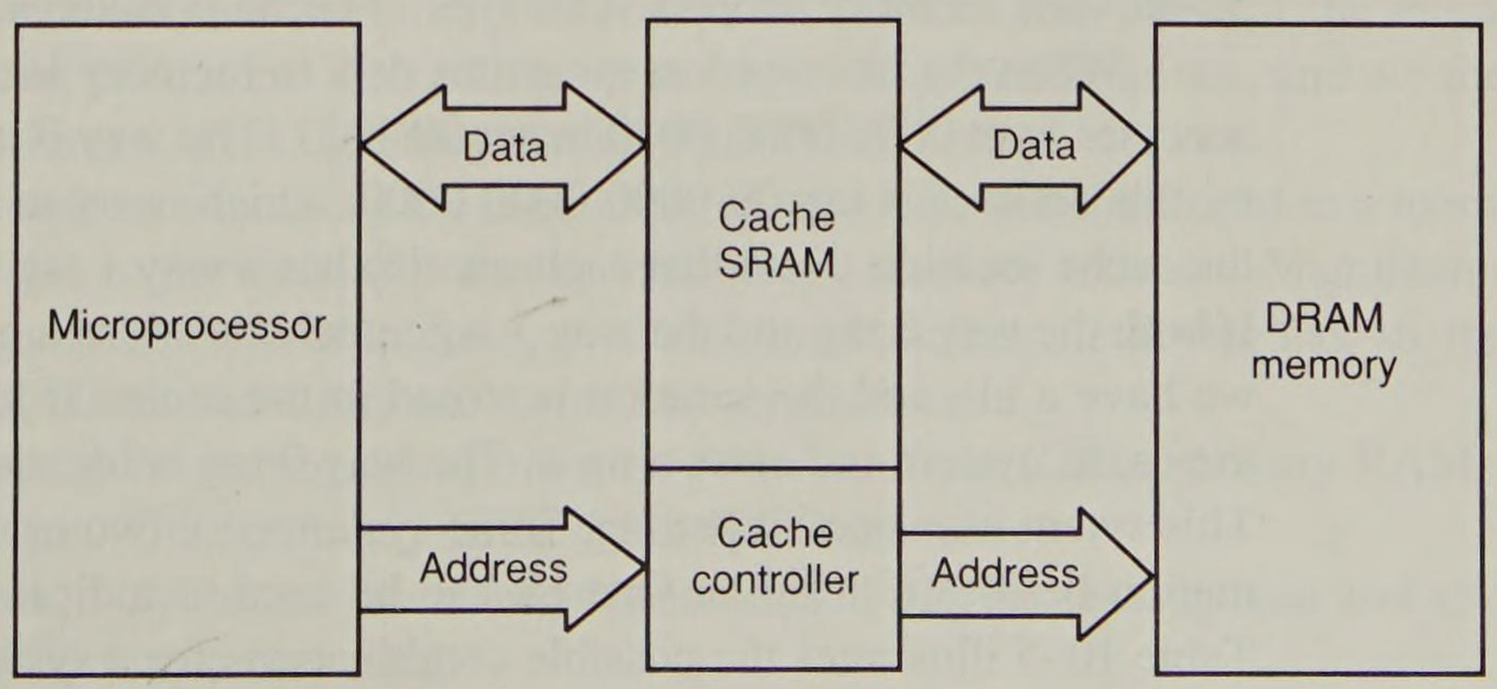



**FIGURE 10–46** The basic format of a cache memory system.

The microprocessor sees the cache memory as if it were the normal memory system. The difference is that the cache issues very few wait states. The main DRAM memory sees the cache controller as the microprocessor. For a microprocessor such as the 80486, the cache size can be 32 bits in width and the DRAM memory can be 32, 64, or 128 bits in width. The wider the DRAM, the fewer the number of wait states required when operating with a cache. If the cache reads/writes 128 bits to memory for each access, the number of wait states is reduced because four 32-bit wide data are accessed for each DRAM read/write.

When the microprocessor reads or writes data, the cache controller checks to see whether the data are present within the cache. If the data are cached (a cache *hit*), the cache controller transfers it between the microprocessor and the cache without wait states. If the data are not cached (a cache *miss*), the cache controller issues wait states to the microprocessor while it retrieves the information from the DRAM memory. Most cache systems retrieve from more than one memory location, so anticipated data is available to the microprocessor without any waits. It is fairly common for the cache controller to retrieve 16 bytes for each memory read. Because programs are normally sequential, the next group of instructions are fetched into the cache before they are needed by the microprocessor, reducing the number of waits required to execute most programs.

Cache sizes are usually between 64K and 1M bytes. Tests reveal that the ideal cache size seems to be 256K bytes for most programming applications. Enlarging the cache size yields only minor improvements in overall system speed. The size of the cache is determined by the software executed by the microprocessor.

## Cache Structure

The cache is structured so some SRAMs store data while others store *tags*. The tags are used to specify the main memory location of the cached information. Tags are needed because the cache sizes are much smaller than the main memory. The memory address is broken into sections that are used to address data in the cache, sets, and tags. Figure 10–47 shows the organization of the cache memory SRAM and the address.

The memory address generated by the microprocessor is broken into three sections that address a byte in the cache, address a set (section) of the cache, and address the tag. The tag, which indicates the data are present in the caches, is used to indicate a hit or a miss. If a 256K byte cache is organized with 4K sets, 12 address lines select a byte in a set. Because each set contains 4K bytes, there are 64 different sets of cache memory, which are selected with another 6 bits of the memory address. The tag for this system is 14 bits in width to form a 32-bit wide memory address. This type of cache is called a two-way set associative cache.

When the microprocessor writes data to memory location 0000F000, for example, it accesses byte 0000 0000 0000 in set 00 1111. The way 0 tag (there are 16K tags) accessed by this address is tag 00 0000 0000 0000, which is set to indicate that data are present in the cache location. Note that each set also has a way 1 tag (4K tags per set) that is also set. If both the way 0 tag and the way 1 tag match the address generated by the microprocessor, we have a hit, and the location is stored in the cache. If location 0000F001H is accessed, the cache system indicates a miss. The way 0 tag is identical, but the way 1 tag is cleared. This sytem uses one bit per tag. Some systems use two or more bits to indicate other information besides a hit/miss. One bit can be used to indicate a read hit, another a write hit. Table 10–5 illustrates the possible combinations for a system that contains 2 tag bits.

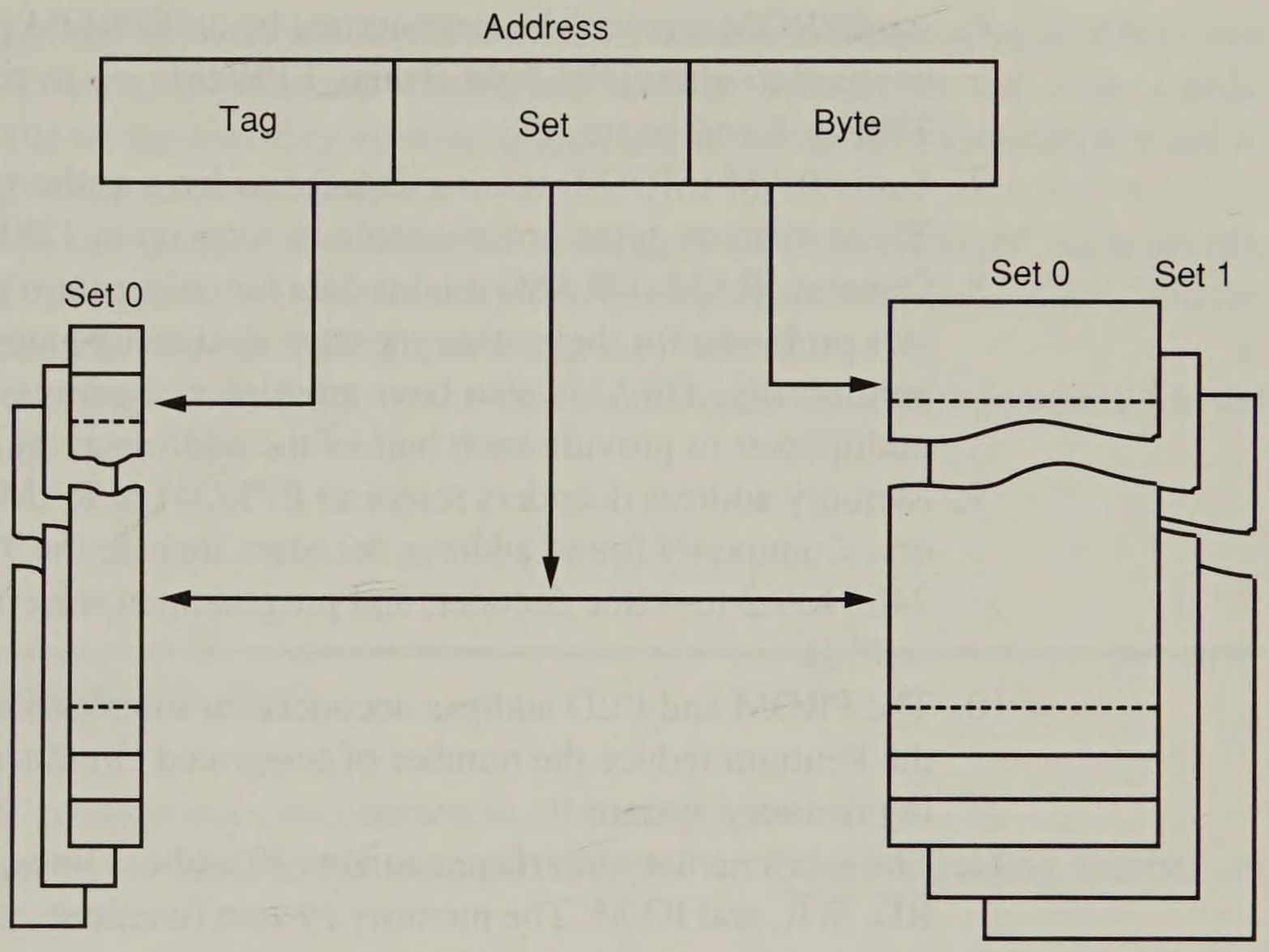



**FIGURE 10–47** The organization of a cache memory. Note that the tags are stored within the controller.

**TABLE 10–5** A cache system that uses 2 tag bits

| *T1* | *T0* | *Function* |
|---|---|---|
| 0 | 0 | Empty |
| 0 | 1 | Read hit |
| 1 | 0 | Write hit |
| 1 | 1 | Invalid |

## 10–10 SUMMARY

1. All memory devices have address inputs; data inputs and outputs, or just outputs; a pin for selection; and one or more pins that control the operation of the memory.
2. Address connections on a memory component are used to select one of the memory locations within the device. Ten address pins have 1,024 combinations, and are therefore able to address 1,024 different memory locations.
3. Data connections on a memory are used to enter information to be stored in a memory location and also to retrieve information read from a memory location. Manufacturers list their memory as, for example, 4K × 4, which means that the device has 4K memory locations (4,096) and 4 bits are stored in each location.
4. Memory selection is accomplished via a chip selection pin ($\overline{CS}$), on many RAMs, or a chip enable pin ($\overline{CE}$), on many EPROM or ROM memories.
5. Memory function is selected by an output enable pin ($\overline{OE}$) for reading data and a write enable pin ($\overline{WE}$) for writing data.

6. An EPROM memory is programmed by an EPROM programmer and can be erased if exposed to ultraviolet light. Today EPROMs are available in sizes from 1K × 8 up to 128K × 8 and larger.
7. Static RAM (SRAM) retains data for as long as the system power supply is attached. These memory types are available in sizes up to 128K × 8.
8. Dynamic RAM (DRAM) retains data for only a short period, usually 2–4 ms. This creates problems for the memory system designer because the DRAM must be refreshed periodically. DRAMs also have multiplexed address inputs that require an external multiplexer to provide each half of the address at the appropriate time.
9. Memory address decoders select an EPROM or RAM at a particular area of the memory. Commonly found address decoders include the 74LS138 3-to-8 line decoder, the 74LS139 2-to-4 line decoder, and programmed selection logic in the form of a PROM or PLD.
10. The PROM and PLD address decoders for microprocessors like the 80386SX through the Pentium reduce the number of integrated circuits required to complete a functioning memory system.
11. The 8-bit memory interface contains 20 address lines, 8 data lines, and 3 control lines: $\overline{RD}$, $\overline{WR}$, and IO/$\overline{M}$. The memory system functions correctly only when all these lines are used for memory interface.
12. The access speed of the EPROM must be compatible with the microprocessor to which it is interfaced. Many EPROMs available today have an access time of 450 ns, which is too slow for almost any modern microprocessor. In order to circumvent this problem, wait states are often inserted to increase the memory access time.
13. Parity checkers are becoming commonplace today in many microprocessor-based microcomputer systems. An extra bit is stored with each byte of memory, making the memory 9 bits wide instead of 8.
14. Error correction features are also available for memory systems, but these require the storage of many more bits. If an 8-bit number is stored with an error correction circuit, 13 bits of memory are required: 5 for an error checking code and 8 for the data. Most error correction integrated circuits are able to correct only a single-bit error.
15. The 80386SX memory interface has a 16-bit data bus and contains a $\overline{MWTC}$ (memory write control) and $\overline{MRDC}$ (memory read control) to select a read or write operation. In addition to these changes, there are the control signals bus high enable ($\overline{BHE}$) and bus low enable ($\overline{BLE}$) that select either the high or low memory banks.
16. The 80386SX memory is organized in two 8-bit banks: high bank and low bank. The high bank of memory is enabled by the $\overline{BHE}$ control signal and the low bank by the $\overline{BLE}$ address signal.
17. Two common schemes for selecting the banks in an 80386SX-based system include (1) a separate decoder for each bank and (2) separate write control signals for each bank with a common decoder.
18. Memory interfaced to the 80386DX and 80486 is 32 bits wide as selected by a 32-bit address bus. Because of the width of this memory, it is organized in four memory banks that are each 8 bits in width. Bank selection signals are provided by the microprocessor as $\overline{BE0}$, $\overline{BE1}$, $\overline{BE2}$, and $\overline{BE3}$.
19. Memory interfaced to the Pentium is 64 bits wide as selected by a 32-bit address bus. Because of the width of this memory it is organized in eight memory banks, each four bits in width. The bank selection signals are provided by the microprocessor.

20. Interleaved memory systems are duplicates of the normal memory system, where one section may contain all the odd words and another section all the even words. Interleaving allows the memory system to look ahead to the next sequential word in the memory to reduce access time.
21. Dynamic RAM controllers are designed to control DRAM memory components. Many DRAM controllers today contain address multiplexers, refresh counters, and the circuitry required to do a periodic DRAM memory refresh.
22. The level 2 cache is an external cache that stores data without wait states for the microprocessor.

## 10–11 QUESTIONS AND PROBLEMS

1. What types of connections are common to all memory devices?
2. List the number of words found in each memory device for the following numbers of address connections:
   (a) 8
   (b) 11
   (c) 12
   (d) 13
3. List the number of data items stored in each of the following memory devices and the number of bits in each datum:
   (a) $2K \times 4$
   (b) $1K \times 1$
   (c) $4K \times 8$
   (d) $16K \times 1$
   (e) $64K \times 4$
4. What is the purpose of the $\overline{CS}$ or $\overline{CE}$ pin on a memory component?
5. What is the purpose of the $\overline{OE}$ pin on a memory device?
6. What is the purpose of the $\overline{WE}$ pin on a RAM?
7. How many words of storage do the following EPROM memory devices contain?
   (a) 2708
   (b) 2716
   (c) 2732
   (d) 2764
   (e) 27128
8. Why won't a 450 ns EPROM work directly with most microprocessors?
9. SRAM is an acronym for what type of device?
10. The 4016 memory has a $\overline{G}$ pin, an $\overline{S}$ pin, and a $\overline{W}$ pin. What are these pins used for in this RAM?
11. How much memory access time is required by the slowest 4016?
12. DRAM is an acronym for what type of device?
13. The 44256 has nine address inputs, yet it is a $256K \times 4$ DRAM. Explain how an 18-bit memory address is forced into nine address inputs.
14. What are the purposes of the $\overline{CAS}$ and $\overline{RAS}$ inputs of a DRAM?

15. How much time is required to refresh the typical DRAM?
16. Why are memory address decoders important?
17. Modify the NAND gate decoder of Figure 10–11 so it selects the memory for address range DF800H–DFFFFH.
18. Modify the NAND gate decoder in Figure 10–11 so it selects the memory for address range 40000H–407FFH.
19. When the $\overline{G1}$ input is high, and $\overline{G2A}$ and $\overline{G2B}$ are both low, what happens to the outputs of the 74LS138 3-to-8 line decoder?
20. Modify the circuit of Figure 10–17 so it addresses memory range 70000H–7FFFFH.
21. Modify the circuit of Figure 10–18 so it addresses memory range 40000H–4FFFFH.
22. Describe the 74LS139 decoder.
23. Why is a PROM address decoder often found in a memory system?
24. Reprogram the PROM in Table 10–1 so it decodes memory address range 80000H–8FFFFH.
25. Reprogram the PROM in Table 10–1 so it decodes memory address range 30000H–3FFFFH.
26. The $\overline{MRCD}$ and $\overline{MWTC}$ control signals are used for what purpose in a memory system interface?
27. Modify the circuit of Figure 10–19 so it selects memory at locations 68000H–6BFFFH.
28. Modify the circuit of Figure 10–19 so it selects eight 2764 8K × 8 EPROMs at memory locations 10000H–1FFFFH.
29. Add another decoder to the circuit of Figure 10–21 so an additional eight 4016 2K × 8 SRAMs are added at locations 10000H–13FFFH.
30. Redesign the main decoder in Figure 10–21 so memory addressing begins at location 80000H.
31. Explain how odd parity is stored in a memory system and how it is checked.
32. The 74LS636 error correction and detection circuit stores a check code with each byte of data. How many bits are stored for the check code?
33. What is the purpose of the SEF pin on the 74LS636?
34. The 74LS636 will correct _____ bits that are in error.
35. Outline the major difference between the buses of the 80386SX and 80386DX microprocessors.
36. What is the purpose of the $\overline{BHE}$ and $\overline{BLE}$ pins on the 80386SX microprocessor?
37. What two methods are used to select the memory?
38. If $\overline{BHE}$ is a logic 0, then the _____ memory bank is selected.
39 If BLE is a logic 0, then the _____ memory bank is selected.
40. Why don't separate bank read strobes need to be developed when interfacing memory to a microprocessor?
41. Modify the circuit of Figure 10–30 so the EPROM is located at memory range C0000H–CFFFFH and the RAM is located at memory range 30000H–33FFFH.
42. Develop a 16-bit wide memory interface that contains SRAM memory at locations 200000H–21FFFFH for the 80386SX microprocessor.
43. Develop a 32-bit wide memory interface that contains EPROM memory at locations FFFF0000H–FFFFFFFFH
44. Develop a 64-bit wide Pentium memory interface that selects EPROM memory at locations 80000000H–8007FFFFH.
45. Why would an interleaved memory system be employed with a microprocessor?

46. Could an interleaved memory system be constructed for the Pentium microprocessor?
47. What is a $\overline{\text{RAS}}$-only cycle?
48. Can DRAM be refreshed while other sections of the memory operate?
49. If a 16K × 1 DRAM requires 2 ms for a refresh and has 128 rows to be refreshed, no more than _____ of time must pass before another row is refreshed.
50. Where is the memory address applied to the TMS4500A DRAM controller?
51. What is the purpose of the REN1 pin on the TMS4500A?
52. What is normally connected to the $\overline{\text{ACW}}$ pin of the TMS4500A?
53. What operating condition on the TMS4500A is TWST used to select?
54. Modify the circuit of Figure 10–42 so it selects memory range 40000H–5FFFFH.
55. Modify the PAL program in Example 10–11 so memory is decoded at addresses 80000000H–87FFFFFFH
56. The address for a cache memory system is broken into sections that access what information?
57. What is a tag in a cache memory?

# CHAPTER 11

# Basic I/O Interface

## INTRODUCTION

A microprocessor is great at solving problems, but if it can't communicate with the outside world, it is of little worth. This chapter outlines some of the basic methods of communications, both serial and parallel, between humans or machines and the microprocessor.

In this chapter, we introduce the basic I/O interface and discuss decoding for I/O devices. Then we provide detail on parallel and serial interfacing, both of which have a wide variety of applications. As applications, we connect analog-to-digital and digital-to-analog converters as well as both DC and stepper motors to the microprocessor.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Explain the operation of the basic input and output interfaces.
2. Decode an 8-, 16-, and 32-bit I/O device so it can be used at any I/O port address.
3. Define handshaking and explain how to use it with I/O devices.
4. Interface and program the 8255 programmable parallel interface.
5. Interface and program the 8279 programmable keyboard/display controller.
6. Interface and program the 8251A serial communications interface adapter.
7. Interface and program the 8254 programmable interval timer.
8. Interface an analog-to-digital converter and a digital-to-analog converter to the microprocessor.
9. Interface both DC and stepper motors to the microprocessor.

## 11–1 INTRODUCTION TO I/O INTERFACE

In this section, we explain the operation of the I/O instructions IN, INS, OUT, and OUTS. We also explain the concept of isolated I/O (sometimes called direct I/O or mapped I/O)

and memory-mapped I/O, the basic input and output interfaces, and handshaking. A working knowledge of these topics will make it easier to understand the connection and operation of the programmable interface components and I/O techniques presented in the remainder of this chapter and text.

## I/O Instructions

The instruction set contains one type of instruction that transfers information to an I/O device (OUT) and another type of instruction to read information from an I/O device (IN). Instructions (INS and OUTS found on all versions except the 8086/8088) are also provided to transfer strings of data between the memory and an I/O device. Table 11–1 lists all versions of each instruction found in the microprocessor's instruction set.

Both the IN and OUT instructions transfer data between an I/O device and the microprocessor's accumulator (AL, AX, or EAX). The I/O address is stored in register DX as a 16-bit I/O address or in the byte immediately following the opcode (p8) as an 8-bit I/O address. Intel calls the 8-bit form (p8) a *fixed address* because it is stored with the instruction, usually in a ROM. The 16-bit I/O address in DX is called a *variable address* because it is stored in a register, which can be varied. Both the INS and OUTS instructions use a variable I/O address contained in the DX register.

Whenever data are transferred using the IN or OUT instruction, the I/O address, often called a *port number*, appears on the address bus. The external I/O interface decodes it in the same manner that it decodes a memory address. The 8-bit fixed port number (p8) appears on address bus connections A7–A0 with bits A15–A8 = 0000 0000. The address connections above A15 are undefined for an I/O instruction. The 16-bit variable port number (DX) appears on address connections A15–A0. This means that the first 256 I/O port addresses (00H–FFH) are accessed by both the fixed and variable I/O instructions, but any I/O address from 0100H–FFFFH is only accessed by the variable I/O address. In many dedicated task systems only the right-most 8 bits of the address are decoded, thus reducing the amount of circuitry required for decoding. In a PC computer, all 16 address bus bits are decoded, with locations 00XXH–03XXH being used for most I/O inside the PC.

The INS and OUTS instructions address the I/O device using the DX register, but do not transfer data between the accumulator and the I/O device as IN and OUT. Instead these instructions transfer data between memory and the I/O device. The memory address is located by ES:DI for the INS instruction and DS:SI for the OUTS instruction. As with other string instructions, the contents of the pointers are incremented or decremented as dictated by the state of the direction flag (DF). Both INS and OUTS can also be prefixed with the REP prefix allowing more than one byte or word to be transferred between I/O and memory.

## Isolated and Memory-Mapped I/O

There are two completely different methods of interfacing I/O to the microprocessor: *isolated I/O* and *memory-mapped I/O*. In isolated I/O, the IN, INS, OUT, and OUTS instructions transfer data between the microprocessor accumulator or memory and the I/O device. In memory-mapped I/O, any instruction that references memory can accomplish the transfer. Both isolated and memory-mapped I/O are in use, so both are discussed in this text.

**TABLE 11–1** Input/Output instructions

| *Instruction* | *Data Width* | *Comment* |
|---|---|---|
| IN AL,p8 | 8 | Read a byte from I/O address p8 into AL |
| IN AL,DX | 8 | Read a byte from the I/O address indexed by DX into AL |
| IN AX,p8 | 16 | Read a word from I/O address p8 into AX |
| IN AX,DX | 16 | Read a word from the I/O address indexed by DX into AX |
| IN EAX,p8 | 32 | Read a doubleword from I/O address p8 into EAX |
| IN EAX,DX | 32 | Read a doubleword from the I/O address indexed by DX into EAX |
| INSB | 8 | Read a byte from the I/O address indexed by DX and store it in the location indexed by ES:DI then increment/decrement DI by 1 |
| INSW | 16 | Read a word from the I/O address indexed by DX and store it in the location indexed by ES:DI then increment/decrement DI by 2 |
| INSD | 32 | Read a doubleword from the I/O address indexed by DX and store it in the location indexed by ES:DI then increment/decrement DI by 4 |
| OUT p8,AL | 8 | Write a byte from AL into I/O address p8 |
| OUT DX,AL | 8 | Write a byte from AL into the I/O address indexed by DX |
| OUT p8,AX | 16 | Write a word from AX into I/O address p8 |
| OUT DX,AX | 16 | Write a word from AX into the I/O address indexed by DX |
| OUT p8,EAX | 32 | Write a doubleword from EAX into I/O address p8 |
| OUT p8,EAX | 32 | Write a doubleword from EAX into the I/O address indexed by DX |
| OUTSB | 8 | Write a byte from the memory location indexed by DS:SI into the I/O address indexed by DX then increment/decrement SI by 1 |
| OUTSW | 16 | Write a word from the memory location indexed by DS:SI into the I/O address indexed by DX then increment/decrement SI by 2 |
| OUTSD | 32 | Write a doubleword from the memory location indexed by DS:SI into the I/O address indexed by DX then increment/decrement SI by 4 |

*Isolated I/O.* The most common I/O transfer technique used in the Intel microprocessor-based system is isolated I/O. The term isolated describes how the I/O locations are isolated from the memory system in a separate I/O address space. (Figure 11–1 illustrates both the isolated and memory-mapped address spaces for any Intel 80X86 or Pentium microprocessor.) The addresses for isolated I/O devices, called *ports*, are separate from the memory. As a result, the user can expand the memory to its full size without using any of the space for I/O devices. A disadvantage of isolated I/O is that the data transferred between I/O and the microprocessor must be accessed by the IN, INS, OUT, and OUTS instructions. Separate control signals for the I/O space are developed (using M/$\overline{IO}$ and W/$\overline{R}$) that indicate an I/O read ($\overline{IORC}$) or an I/O write ($\overline{IOWC}$) operation. These signals indicate that an I/O port address appears on the address bus used to select the I/O device. In the personal computer, isolated I/O ports are used for controlling peripheral devices. As a rule, an 8-bit port address is used to access devices located on the system board, such as

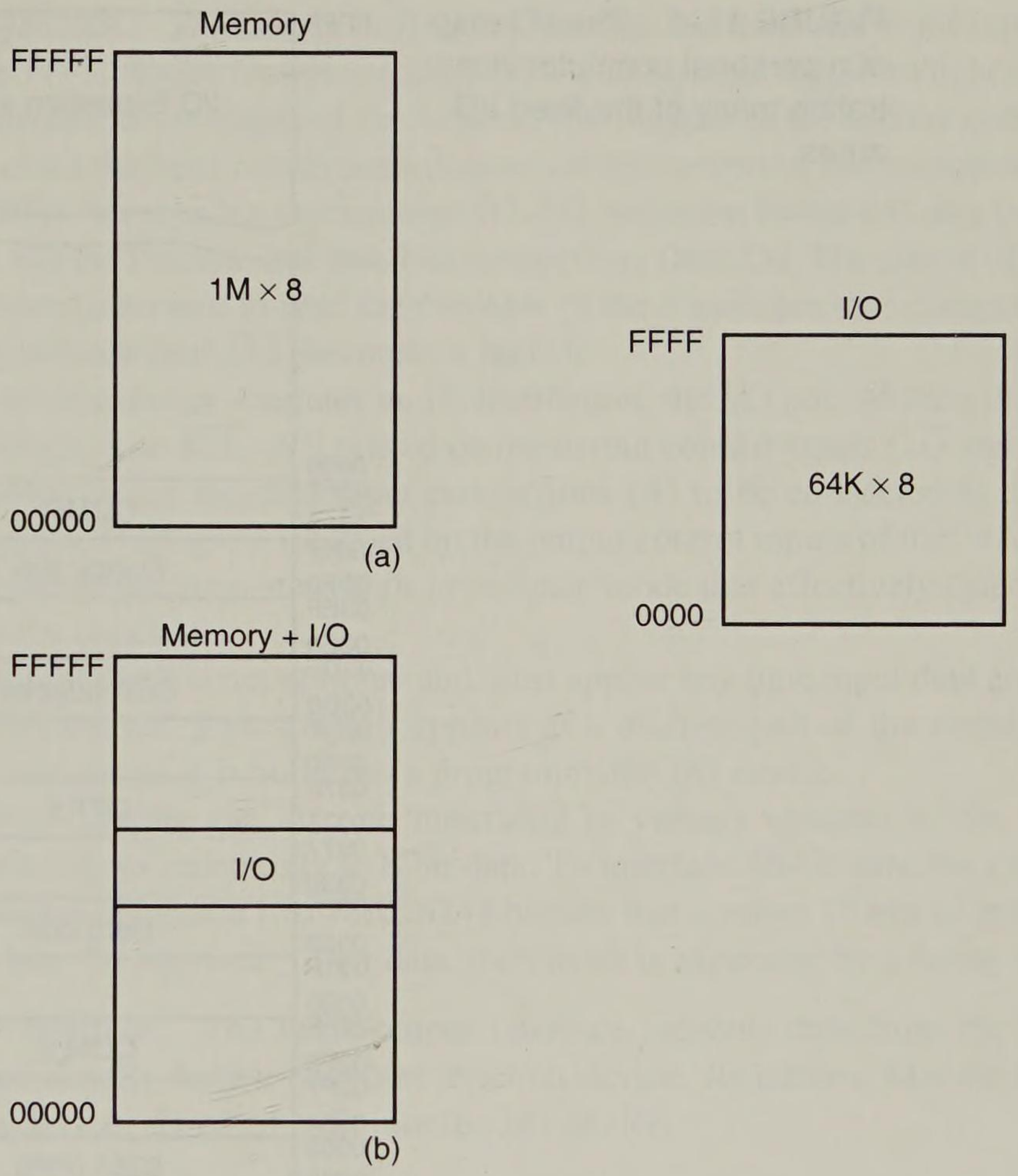



**FIGURE 11–1** The memory and I/O maps for the 8086/8088 microprocessor. (a) Isolated I/O. (b) Memory-mapped I/O.

the timer and keyboard interface, and a 16-bit port address is used to access serial and parallel ports as well as video and disk drive systems.

*Memory-Mapped I/O.* Unlike isolated I/O, memory-mapped I/O does not use the IN, INS, OUT, or OUTS instructions. Instead, it uses any instruction that transfers data between the microprocessor and memory. A memory-mapped I/O device is treated as a memory location in the memory map. The main advantage of memory-mapped I/O is that any memory transfer instruction can be used to access the I/O device. The main disadvantage is that a portion of the memory system is used as the I/O map. This reduces the amount of memory available to applications. Another advantage is that the $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ signals have no function in a memory-mapped I/O system, which may reduce the amount of circuitry required for decoding.

## Personal Computer I/O Map

The personal computer uses part of the I/O map for dedicated functions. Figure 11–2 shows the I/O map for the PC. Note that I/O space between ports 0000H and 03FFH are normally reserved for the computer system. The I/O ports located at 0400H–FFFFH are generally available for user applications. Note that the 80287 arithmetic coprocessor uses

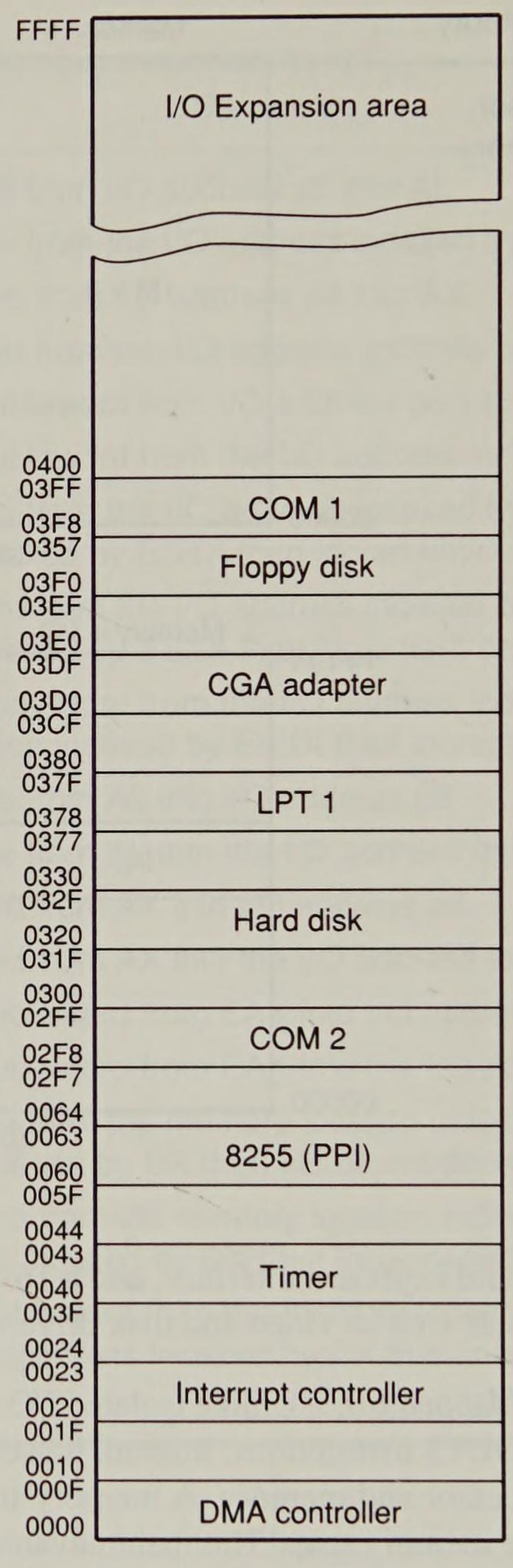



**FIGURE 11–2** The I/O map of a personal computer illustrating many of the fixed I/O areas.

I/O addresses 00F8–00FD for communications. For this reason Intel reserves I/O ports 00F0H–00FFH. The 80386–Pentium uses I/O ports 800000F8–800000FDH for communications to their coprocessors. The I/O ports located between 0000H and 00FFH are accessed via the fixed port I/O instructions, and the ports located above 00FFH are accessed via the variable I/O port instructions.

## Basic Input and Output Interfaces

The basic input device is a set of three-state buffers. The basic output device is a set of data latches. The term IN refers to moving data from the I/O device into the microprocessor and the term OUT refers to moving data out of the microprocessor to the I/O device.

*The Basic Input Interface.* Three-state buffers are used to construct the 8-bit input port depicted in Figure 11–3. Notice that the external TTL data (simple toggle switches in this example) are connected to the inputs of the buffers. The outputs of the buffers connect to the data bus. The exact data bus connections depend on the version of the microprocessor. For example, the 8088 has data bus connections D7–D0, while the 80486 has data bus connections D31–D0 and the Pentium has data bus connections D63–D0. The circuit of Figure 11–3 allows the microprocessor to read the contents of the 8 switches that connect to the data bus when the select signal $\overline{\text{SEL}}$ becomes a logic 0.

When the microprocessor executes an IN instruction, the I/O port address is decoded to generate the logic 0 on $\overline{\text{SEL}}$. A 0 placed on the output control inputs ($\overline{1G}$ and $\overline{2G}$) of the 74ALS244 buffer causes the data input connections (A) to be connected to the data output (Y) connections. If a logic 1 is placed on the output control inputs of the 74ALS244 buffer, the device enters the three-state high-impedance mode that effectively disconnects the switches from the data bus.

This basic input circuit is not optional and must appear any time input data are interfaced to the microprocessor. Sometimes it appears as a discrete part of the circuit, as in Figure 11–3, and sometimes it is built into a programmable I/O device.

Sixteen- or 32-bit data can also be interfaced to various versions of the microprocessor, but not nearly as commonly as 8-bit data. To interface 16-bit data, the circuit in Figure 11–3 is doubled to include two 74ALS244 buffers that connect 16 bits of input data to the 16-bit data bus. To interface 32-bit data, the circuit is expanded by a factor of 4.

*The Basic Output Interface.* The basic output interface receives data from the microprocessor and must usually hold it for some external device. Its latches, like the buffers found in the input device, are often built into the I/O device.

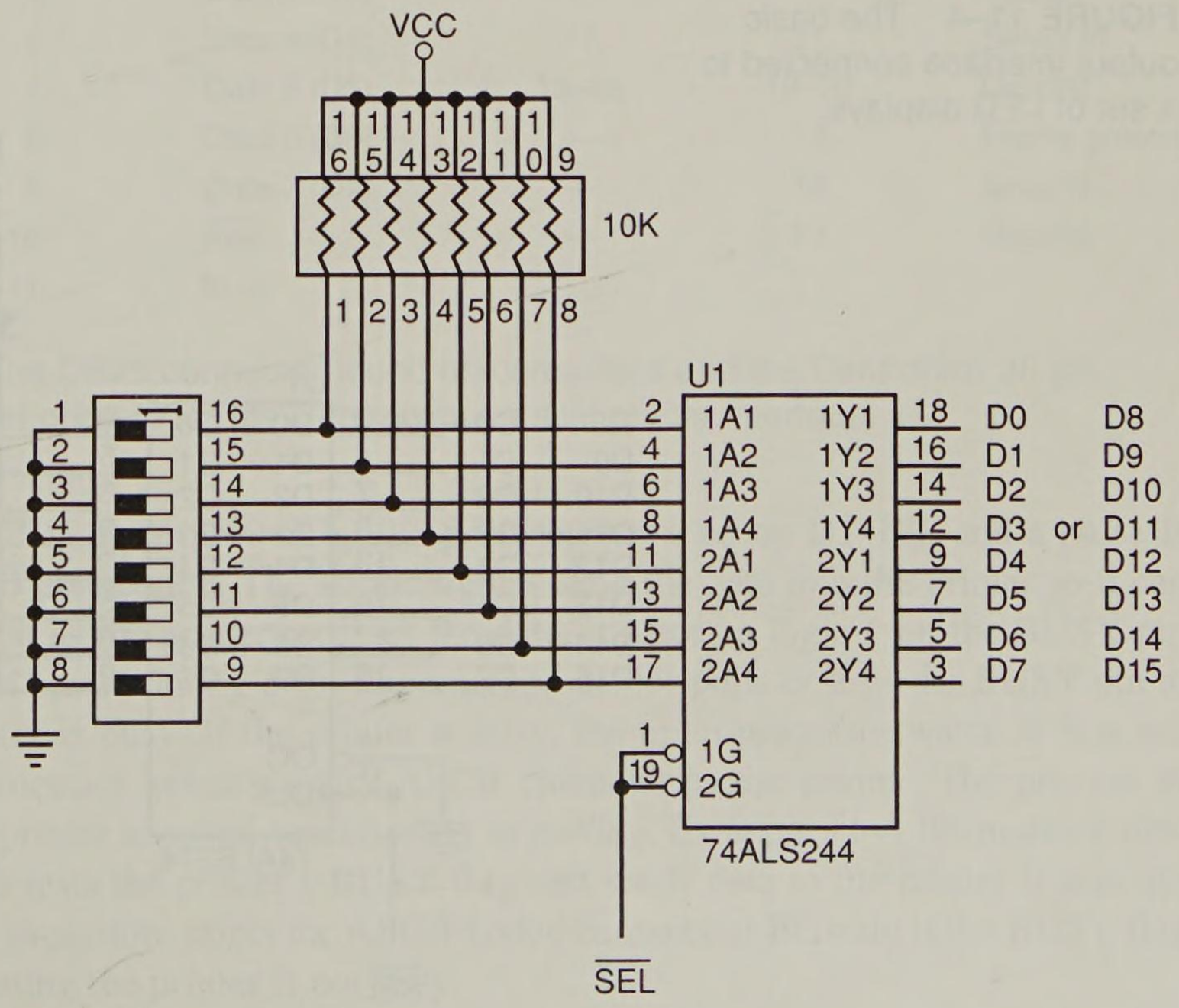



**FIGURE 11–3** The basic input interface illustrating the connection of eight switches. Note that the 74ALS244 is a three-state buffer that controls the application of the switch data to the data bus.

Figure 11–4 shows how 8 simple light-emitting diodes (LEDs) connect to the microprocessor through a set of 8 data latches. The latch stores the number output by the microprocessor from the data bus, so the LEDs can be lit with any 8-bit binary number. Latches are needed to hold the data, because when the microprocessor executes an OUT instruction, the data are only present on the data bus for less than 1.0 µs. Without a latch, the viewer would never see the LEDs illuminate.

When the OUT instruction executes, the data from AL, AX, or EAX are transferred to the latch via the data bus. Here the D inputs of a 74ALS374 octal latch are connected to the data bus to capture the output data, and the Q outputs of the latch are attached to the LEDs. When a Q output becomes a logic 0, the LED lights. Each time the OUT instruction executes, the $\overline{\text{SEL}}$ signal to the latch activates, capturing the data output to the latch from the data bus. The data are held until the next OUT instruction executes.

## Handshaking

Many I/O devices accept or release information at a much slower rate than the microprocessor. Another method of I/O control, called *handshaking* or *polling*, synchronizes the I/O device with the microprocessor. An example device that requires handshaking is a parallel printer that prints 100 characters per second (CPS). It is obvious that the microprocessor can definitely send more than 100 CPS to the printer, so a way must be developed to slow the microprocessor down to match speeds with the printer.

Figure 11–5 illustrates the typical input and output connections found on a printer. Here data are transferred through a series of data connections (D7–D0). BUSY indicates that the printer is busy, and $\overline{\text{STB}}$ is a clock pulse used to send data into the printer for printing.

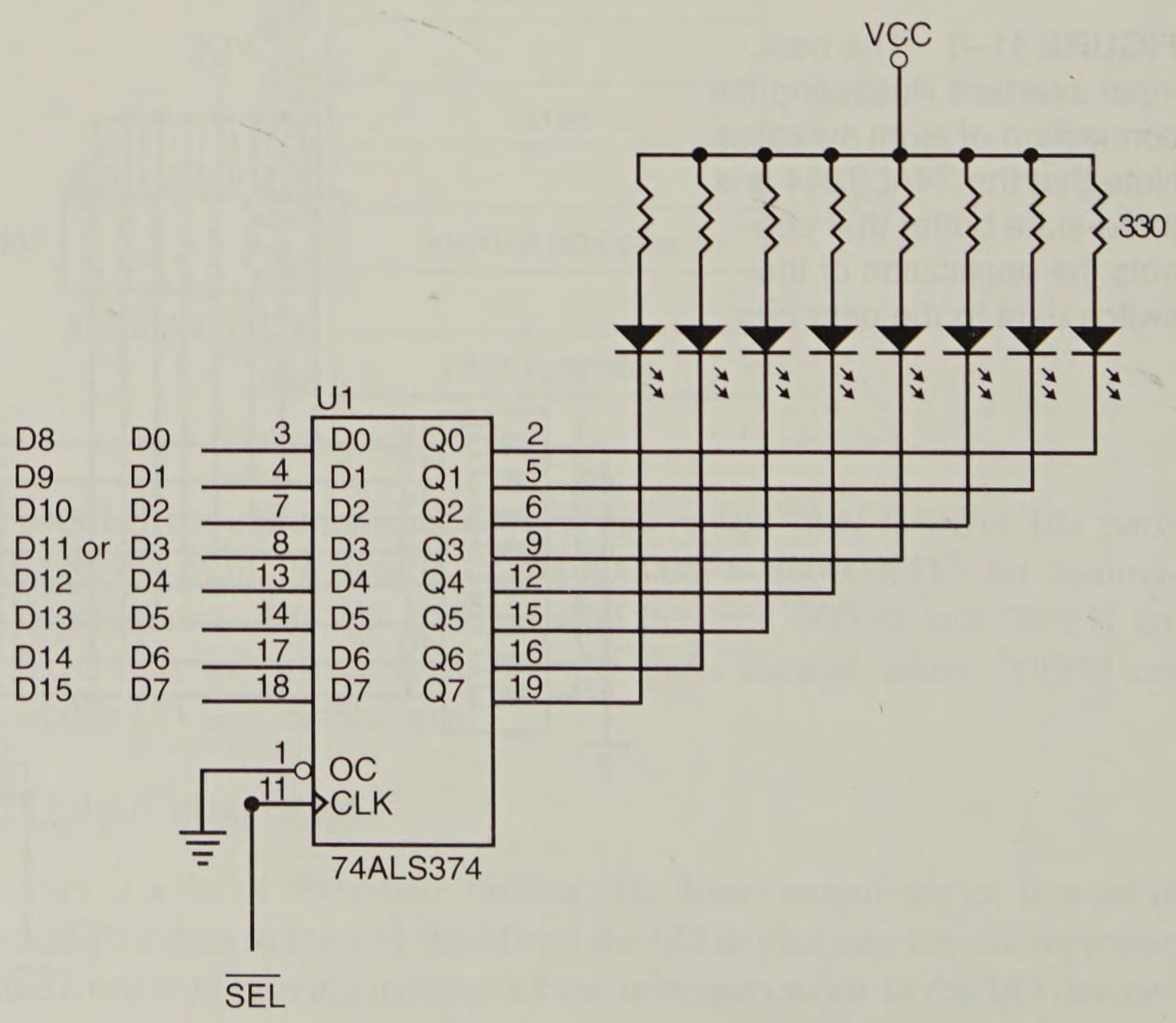
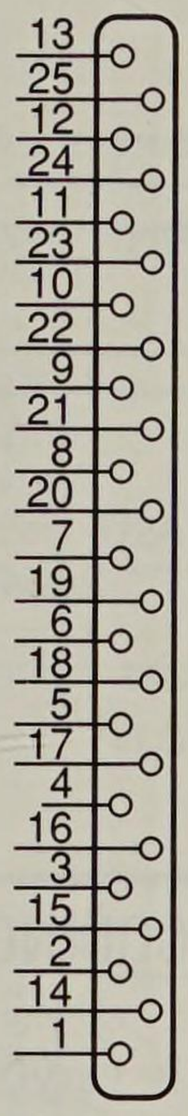
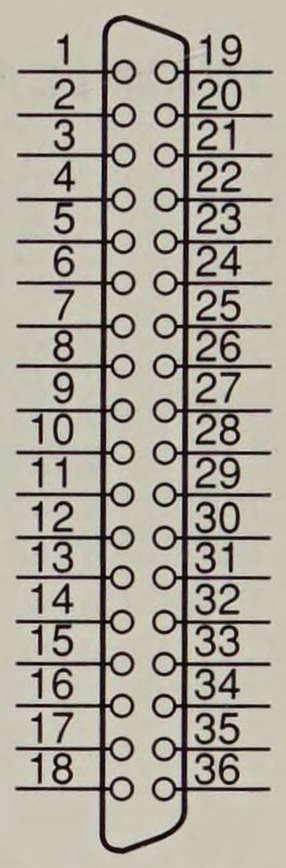



**FIGURE 11–4** The basic output interface connected to a set of LED displays.




Connector DB25




Connector CENT36

| DB25 Pin number | CENT36 Pin number | Function | DB25 Pin number | CENT36 Pin number | Function |
|---|---|---|---|---|---|
| 1 | 1 | $\overline{\text{Data Strobe}}$ | 12 | 12 | Paper empty |
| 2 | 2 | Data 0 (D0) | 13 | 13 | Select |
| 3 | 3 | Data 1 (D1) | 14 | 14 | Afd |
| 4 | 4 | Data 2 (D2) | 15 | 32 | $\overline{\text{Error}}$ |
| 5 | 5 | Data 3 (D3) | 16 | — | $\overline{\text{RESET}}$ |
| 6 | 6 | Data 4 (D4) | 17 | 31 | Select in |
| 7 | 7 | Data 5 (D5) | 18–25 | 19–30 | Ground |
| 8 | 8 | Data 6 (D6) | — | 17 | Frame ground |
| 9 | 9 | Data 7 (D7) | — | 16 | Ground |
| 10 | 10 | $\overline{\text{Ack}}$ | — | 33 | Ground |
| 11 | 11 | Busy | | | |

**FIGURE 11–5** The DB25 connector found on computers and the Centronics 36-pin connector found on printers for the Centronics parallel printer interface.

The ASCII data to be printed by the printer are placed on D7–D0, and a pulse is applied to the $\overline{\text{STB}}$ connection. The strobe signals send the data into the printer so it can be printed. As soon as the printer receives the data, it places a logic 1 on the BUSY pin indicating that it is busy printing data. The microprocessor polls or tests the BUSY pin to decide if the printer is busy. If the printer is busy, the microprocessor waits; if it is not busy, the microprocessor sends another ASCII character to the printer. The process of *interrogating* the printer is called *handshaking* or *polling*. Example 11–1 illustrates a simple procedure that tests the printer's BUSY flag and sends data to the printer if it is not busy. The PRINT procedure prints the ASCII-coded contents of BL only if the BUSY flag is a logic 0, indicating the printer is not busy.

**EXAMPLE 11–1**

```
                        ;procedure that prints the contents of BL
                        ;
0000                    PRINT  PROC FAR

0000 E4 4B                     IN   AL,BUSY          ;get BUSY flag
0002 A8 04                     TEST AL,BUSY_BIT      ;test BUSY bit
0004 75 FA                     JNE  PRINT            ;if BUSY = 1
0006 8A C3                     MOV  AL,BL            ;print BL
0008 E6 4A                     OUT  PRINTER,AL
000A CB                        RET

000B                    PRINT  ENDP
```

## 11–2 I/O PORT ADDRESS DECODING

I/O port address decoding is very similar to memory address decoding, especially for memory-mapped I/O devices. In fact, we do not discuss memory-mapped I/O decoding because it is treated exactly the same as memory, except that $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ are not used, since there is no IN or OUT instruction. The decision to use memory-mapped I/O is often determined by the size of the memory system and the placement of the I/O devices in the system.

The main difference between memory decoding and isolated I/O decoding is the number of address pins connected to the decoder. We decode A32–A0, A23–A0, or A19–A0 for memory and A15–A0 for isolated I/O. Sometimes, if the I/O devices use only fixed I/O addressing, we decode only A7–A0. Another difference is that we use $\overline{\text{IORC}}$ and $\overline{\text{IOWC}}$ to activate I/O devices for a read or a write operation. On earlier versions of the microprocessor, IO/$\overline{\text{M}}$ = 1 and $\overline{\text{RD}}$ or $\overline{\text{WR}}$ are used to activate I/O devices.

### Decoding 8-Bit I/O Addresses

As mentioned, the fixed I/O instruction uses an 8-bit I/O port address that appears on A15–A0 as 0000H–00FFH. If a system contains less than 256 I/O devices we often decode only address connections A7–A0 for an 8-bit I/O port address. Please note that the DX register can also address I/O ports 00H–FFH. Also note that if the address is decoded as an 8-bit address, then we can never include I/O devices that use a 16-bit I/O address.

Figure 11–6 illustrates a 74ALS138 decoder that decodes 8-bit I/O ports F0H through F7H. (We assume that the system will only use I/O ports 00H–FFH for this decoder.) This decoder is identical to a memory address decoder except we only connect address bits A7–A0 to the inputs of the decoder. Figure 11–7 shows the PAL version of this decoder. Notice that this is a better decoder circuit, because the number of integrated circuits has been reduced to one device, the PAL. The program for the PAL appears in Example 11–2.

**EXAMPLE 11–2**

```
TITLE           Address Decoder
PATTERN         Test 2
REVISION        A
```

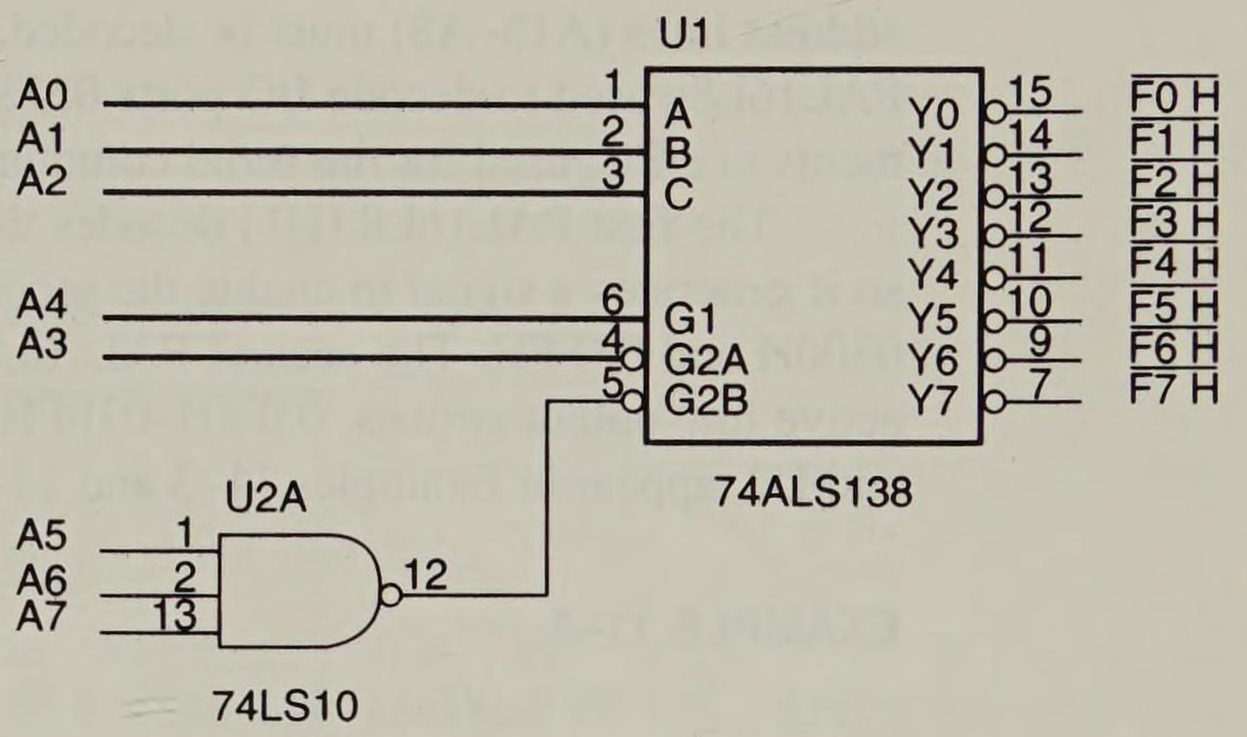



**FIGURE 11–6** A port decoder that decodes 8-bit I/O ports. This decoder generates active low outputs for ports F0H–F7H.

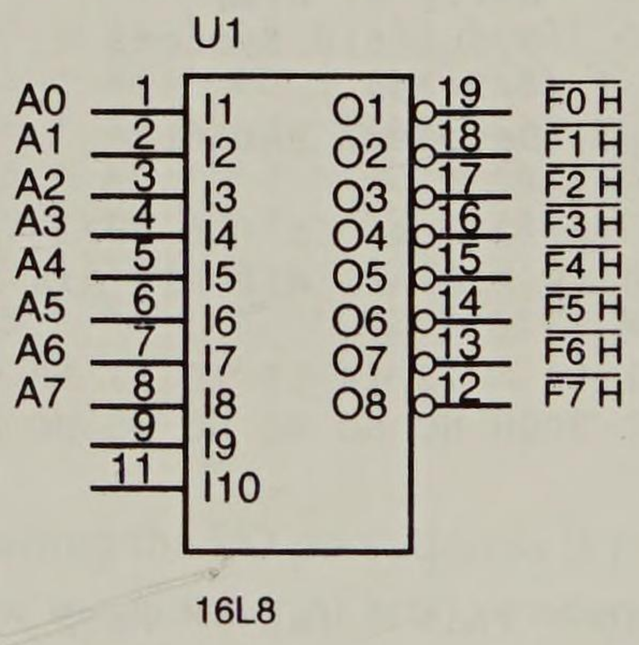



**FIGURE 11–7** A PAL16L8 decoder that generates I/O port signals for ports F0H–F7H.

```
AUTHOR          Barry B. Brey
COMPANY         Symbiotic Systems
DATE            4/23/94
CHIP            Decoder2 PAL16L8

;pins     1  2  3  4  5  6  7  8  9  10
          A0 A1 A2 A3 A4 A5 A6 A7 NC GND

;pins     11 12 13 14 15 16 17 18 19 20
          NC F7 F6 F5 F4 F3 F2 F1 F0 VCC

EQUATIONS

/F0 = A7 * A6 * A5 * A4 * A3 * /A2 * /A1 * /A0
/F1 = A7 * A6 * A5 * A4 * A3 * /A2 * /A1 * A0
/F2 = A7 * A6 * A5 * A4 * A3 * /A2 * A1 * /A0
/F3 = A7 * A6 * A5 * A4 * A3 * /A2 * A1 * A0
/F4 = A7 * A6 * A5 * A4 * A3 * A2 * /A1 * /A0
/F5 = A7 * A6 * A5 * A4 * A3 * A2 * /A1 * A0
/F6 = A7 * A6 * A5 * A4 * A3 * A2 * A1 * /A0
/F7 = A7 * A6 * A5 * A4 * A3 * A2 * A1 * A0
```

## Decoding 16-Bit I/O Addresses

We also decode 16-bit I/O addresses, especially in personal computer systems. The main difference between decoding an 8-bit I/O address and a 16-bit I/O address is that 8 additional

address lines (A15–A8) must be decoded. Figure 11–8 illustrates a circuit that contains 2 PAL16L8s used to decode I/O ports 03F8H–03FFH. These are common I/O port assignments in a PC, used for the serial communications port.

The first PAL16L8 (U1) decodes the first 8 bits of the I/O port address (A15–A8), so it generates a signal to enable the second PAL16L8 (U2) for any I/O address between 0300H and 03FFH. The second PAL16L8 further decodes the I/O address to produce 8 active low-output strobes, $\overline{03F8H}$–$\overline{03FFH}$. The programs for both PAL16L8 devices, U1 and U2, appear in Examples 11–3 and 11–4.

**EXAMPLE 11–3**

```
TITLE          Address Decoder
PATTERN        Test 3
REVISION       A
AUTHOR         Barry B. Brey
COMPANY        Symbiotic Systems
DATE           4/24/94
CHIP           Decoder3 PAL16L8

;pins     1  2  3   4   5   6   7   8   9  10
          A8 A9 A10 A11 A12 A13 A14 A15 NC GND

;pins     11 12   13 14 15 16 17 18 19 20
          NC 300H NC NC NC NC NC NC NC VCC

EQUATIONS

/300H = /A15 * /A14 * /A13 * /a12 * /A11 * /A10 * /A9 * /A8
```

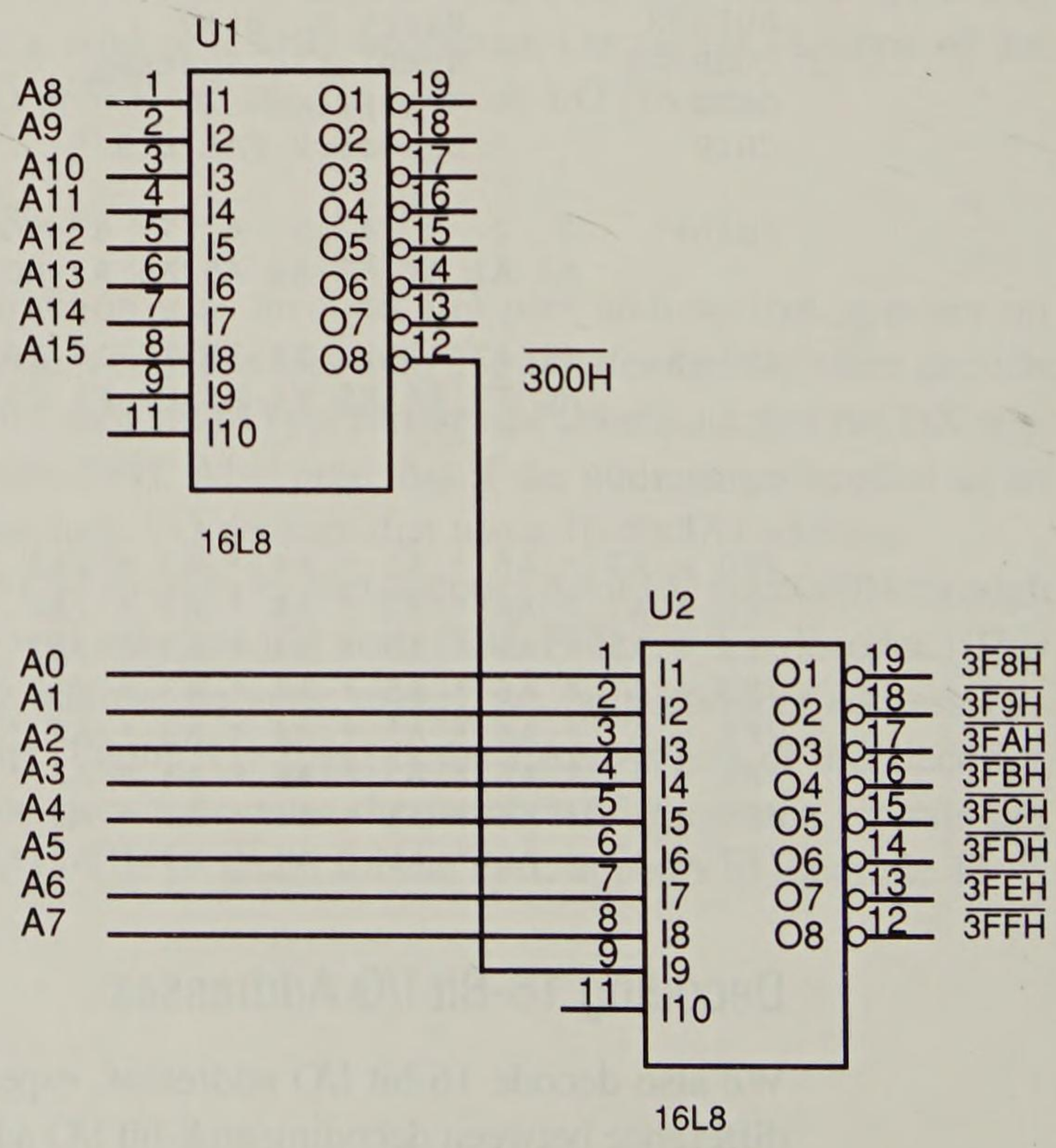



**FIGURE 11–8** A PAL16L8 circuit that decodes 16-bit I/O ports 3F8H–3FFH.

**EXAMPLE 11–4**

```
TITLE          Address Decoder
PATTERN        Test 4
REVISION       A
AUTHOR         Barry B. Brey
COMPANY        Symbiotic Systems
DATE           4/25/94
CHIP           Decoder4 PAL16L8

;pins     1  2  3  4  5  6  7  8  9    10
          A0 A1 A2 A3 A4 A5 A6 A7 300H GND

;pins     11 12   13   14   15   16   17   18   19   20
          NC 3FFH 3FEH 3FDH 3FCH 3FBH 3FAH 3F9H 3F8H VCC
EQUATIONS

/3F8H = /300H * A7 * A6 * A5 * A4 * A3 * /A2 * /A1 * /A0
/3F9H = /300H * A7 * A6 * A5 * A4 * A3 * /A2 * /A1 * A0
/3FAH = /300H * A7 * A6 * A5 * A4 * A3 * /A2 * A1 * /A0
/3FBH = /300H * A7 * A6 * A5 * A4 * A3 * /A2 * A1 * A0
/3FCH = /300H * A7 * A6 * A5 * A4 * A3 * A2 * /A1 * /A0
/3FDH = /300H * A7 * A6 * A5 * A4 * A3 * A2 * /A1 * A0
/3FEH = /300H * A7 * A6 * A5 * A4 * A3 * A2 * A1 * /A0
/3FFH = /300H * A7 * A6 * A5 * A4 * A3 * A2 * A1 * A0
```

## 8- and 16-Bit I/O Ports

Now that we understand that decoding the I/O port address is probably simpler than decoding a memory address (because of the number of bits), we can explain how data are transferred between the microprocessor and 8- or 16-bit I/O devices. Data transferred to an 8-bit I/O device exists in one of the I/O banks of the 80386SX. The I/O system contains two 8-bit memory banks just as memory does. This is illustrated in Figure 11–9, which shows the separate I/O banks for a 16-bit system, such as the 80386SX.

Because two I/O banks exist, any 8-bit I/O write requires a separate write strobe to function correctly. I/O reads do not require separate read strobes because, as with memory, the microprocessor only reads the byte it expects and ignores the other byte. The only time a read can cause problems is when the I/O device responds incorrectly to a read operation.

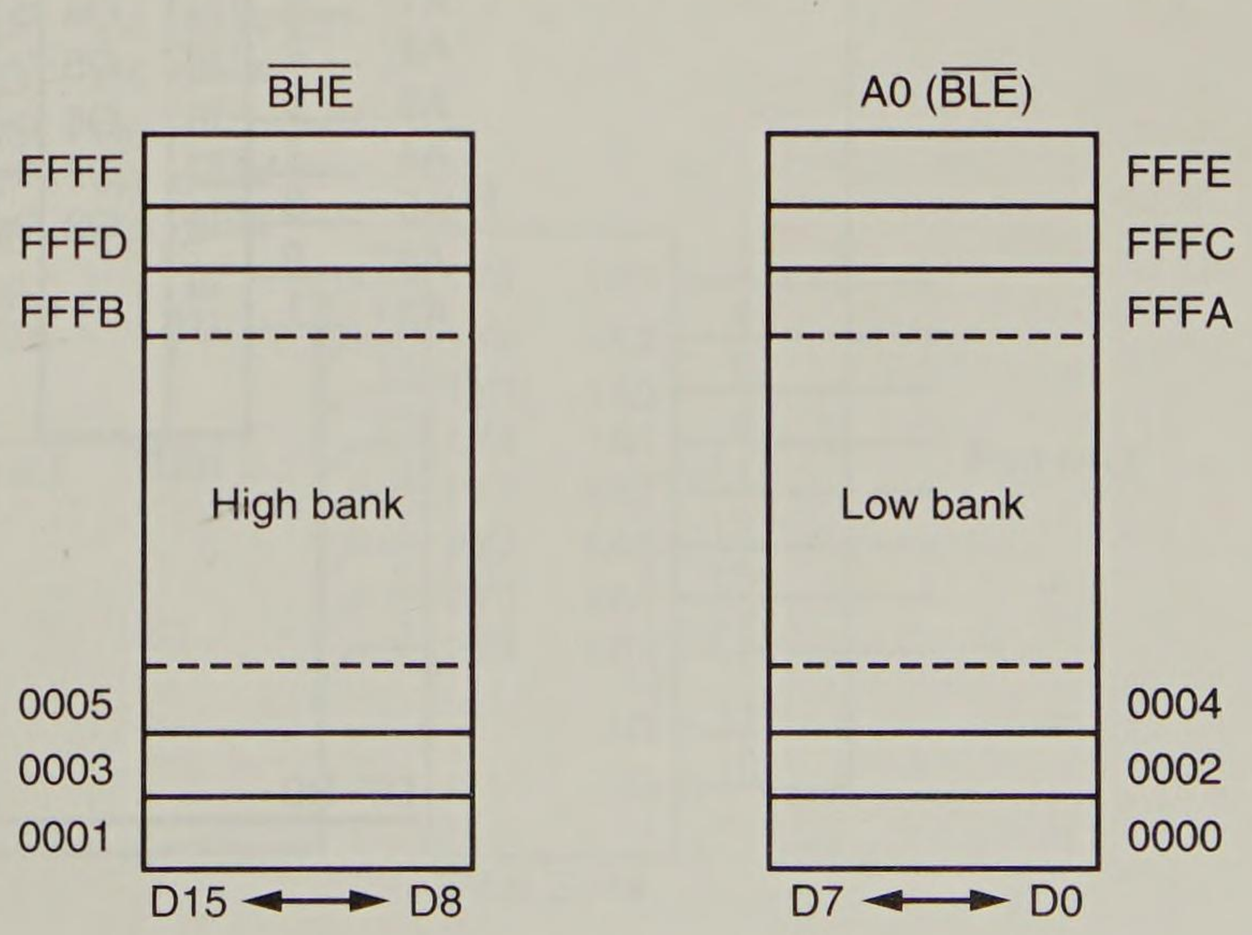



**FIGURE 11–9** The I/O banks found in the 80386SX microprocessor-based system.

In the case of an I/O device that responds to a read from the wrong bank, we may need to include separate read signals. This case is discussed later in this chapter.

Figure 11–10 illustrates a system that contains two different 8-bit output devices located at 8-bit I/O addresses 40H and 41H. Because these are 8-bit devices, and because they appear in different I/O banks, we generate separate I/O write signals. The program for the PAL16L8 decoder used in Figure 11–9 is illustrated in Example 11–5.

**EXAMPLE 11–5**

```
TITLE          Address Decoder
PATTERN        Test 5
REVISION       A
AUTHOR         Barry B. Brey
COMPANY        Symbiotic Systems
DATE           4/26/94
CHIP           Decoder5 PAL16L8

;pins     1   2    3   4  5  6  7  8  9  10
          BHE IOWC BLE A1 A2 A3 A4 A5 A6 GND
```

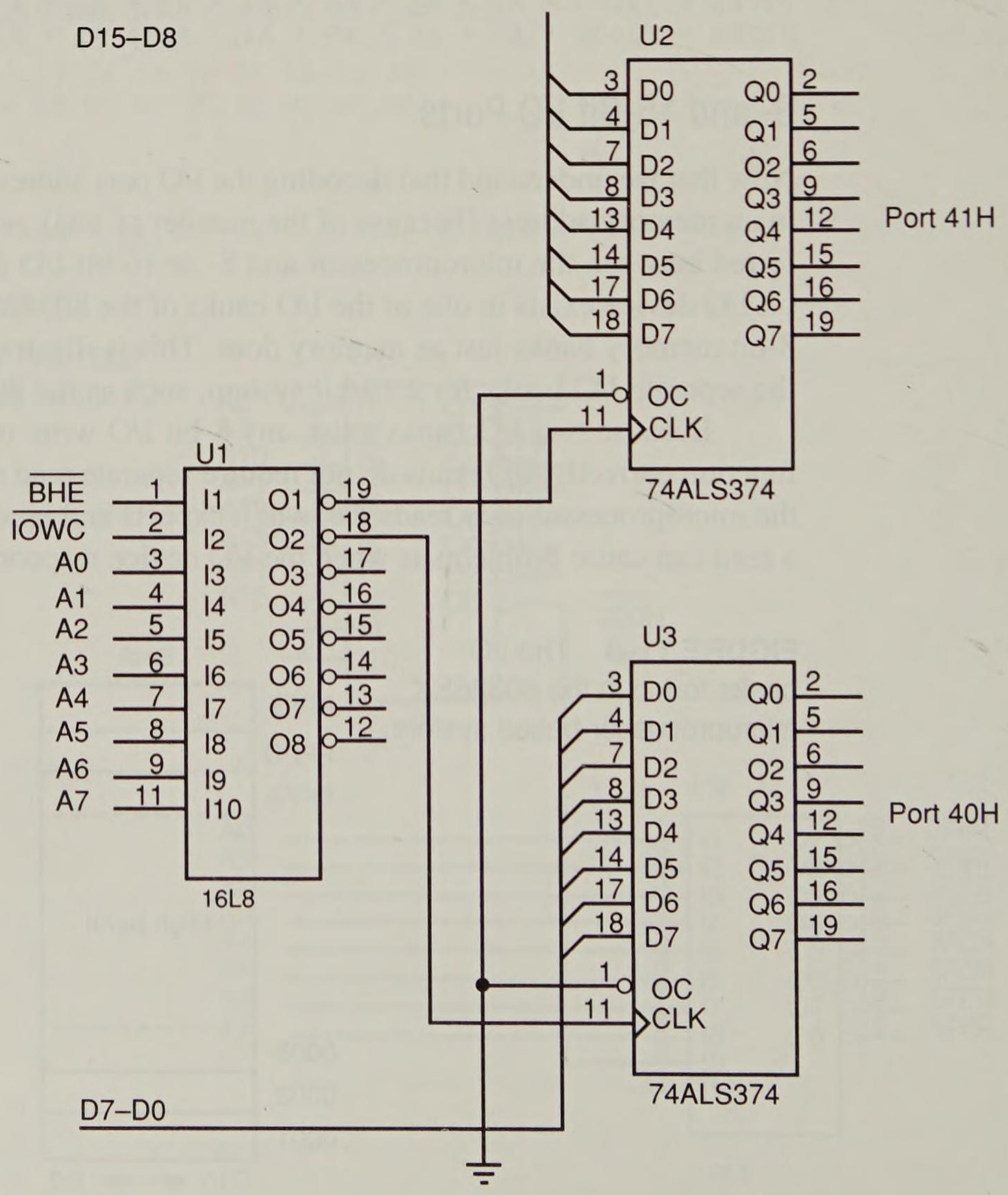



**FIGURE 11–10** An I/O port decoder that selects ports 40H and 41H for output data.

```
;pins     11 12 13 14 15 16 17 18  19  20
          A7 NC NC NC NC NC NC 040 041 VCC
EQUATIONS

/040 = /BLE * /IOWC * /A7 * A6 * /A5 * /A4 * /A3 * /A2 * /A1
/041 = /BHE * /IOWC * /A7 * A6 * /A5 * /A4 * /A3 * /A2 * /A1
```

When selecting 16-bit wide I/O devices, the A0 and $\overline{\text{BHE}}$ pins have no function, because both I/O banks are selected together. Although 16-bit I/O devices are relatively rare, a few do exist for analog-to-digital and digital-to-analog converters, as well as for some video and disk memory interfaces.

Figure 11–11 illustrates a 16-bit input device connected to function at 8-bit I/O addresses 64H and 65H. Notice that the PAL16L8 decoder does not have a connection for address bit A0 and $\overline{\text{BHE}}$, because these signals do not apply to 16-bit-wide I/O devices. The program for the PAL16L8 is illustrated in Example 11–6 to show how the enable signals are generated for the three-state buffers (74ALS244) used as input devices.

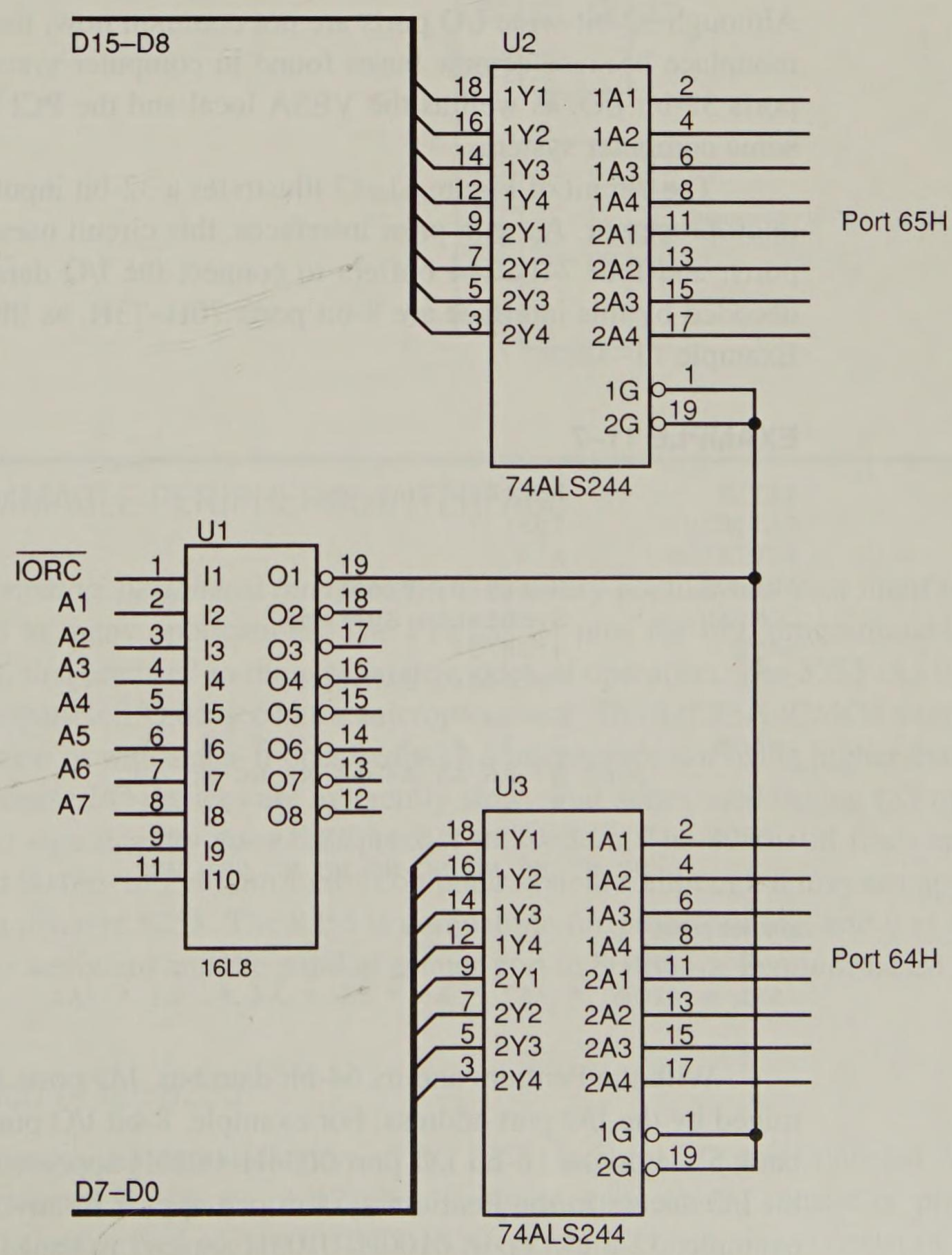



**FIGURE 11–11** A 16-bit I/O port decoded at I/O addresses 64H and 65H.

**EXAMPLE 11–6**

```
TITLE          Address Decoder
PATTERN        Test 6
REVISION       A
AUTHOR         Barry B. Brey
COMPANY        Symbiotic Systems
DATE           4/27/94
CHIP           Decoder5 PAL16L8

;pins     1    2  3  4  5  6  7  8  9  10
          IORC A1 A2 A3 A4 A5 A6 A7 NC GND

;pins     11 12 13 14 15 16 17 18 19  20
          NC NC NC NC NC NC NC NC 06X VCC
EQUATIONS

/06X = /IORC * /A7 * A6 * A5 * /A4 * /A3 * /A2 * /A1
```

## 32-Bit-Wide I/O Ports

Although 32-bit-wide I/O ports are not common now, they may eventually become commonplace because of new buses found in computer systems. The EISA system bus supports 32-bit I/O, as well as the VESA local and the PCI bus, but these are found only in some computer systems.

The circuit of Figure 11–12 illustrates a 32-bit input port for the 80386DX or 80486 microprocessor. As with prior interfaces, this circuit uses a single PAL to decode the I/O ports, and four 74LS244 buffers to connect the I/O data to the data bus. The I/O ports decoded by this interface are 8-bit ports 70H–73H, as illustrated by the PAL program in Example 11–7.

**EXAMPLE 11–7**

```
TITLE          Address Decoder
PATTERN        Test 7
REVISION       A
AUTHOR         Barry B. Brey
COMPANY        Symbiotic Systems
DATE           4/28/94
CHIP           Decoder7 PAL16L8

;pins     1    2  3  4  5  6  7  8  9  10
          IORC A7 A6 A5 A4 A3 A2 NC NC GND

;pins     11 12 13 14 15 16 17 18 19  20
          NC NC NC NC NC NC NC NC SEL VCC

EQUATIONS

/SEL = /IORC * /A7 * A6 * A5 * A4 * /A3 * /A2
```

With the Pentium and its 64-bit data bus, I/O ports appear in various banks as determined by the I/O port address. For example, 8-bit I/O port 0034H appears in Pentium I/O bank 5, while the 16-bit I/O port 0034H–0035H appears in Pentium banks 5 and 6. A 32-bit I/O access in the Pentium system can appear in any four consecutive I/O banks. For example, 32-bit I/O port 0100H–0103H appears in banks 0–3.

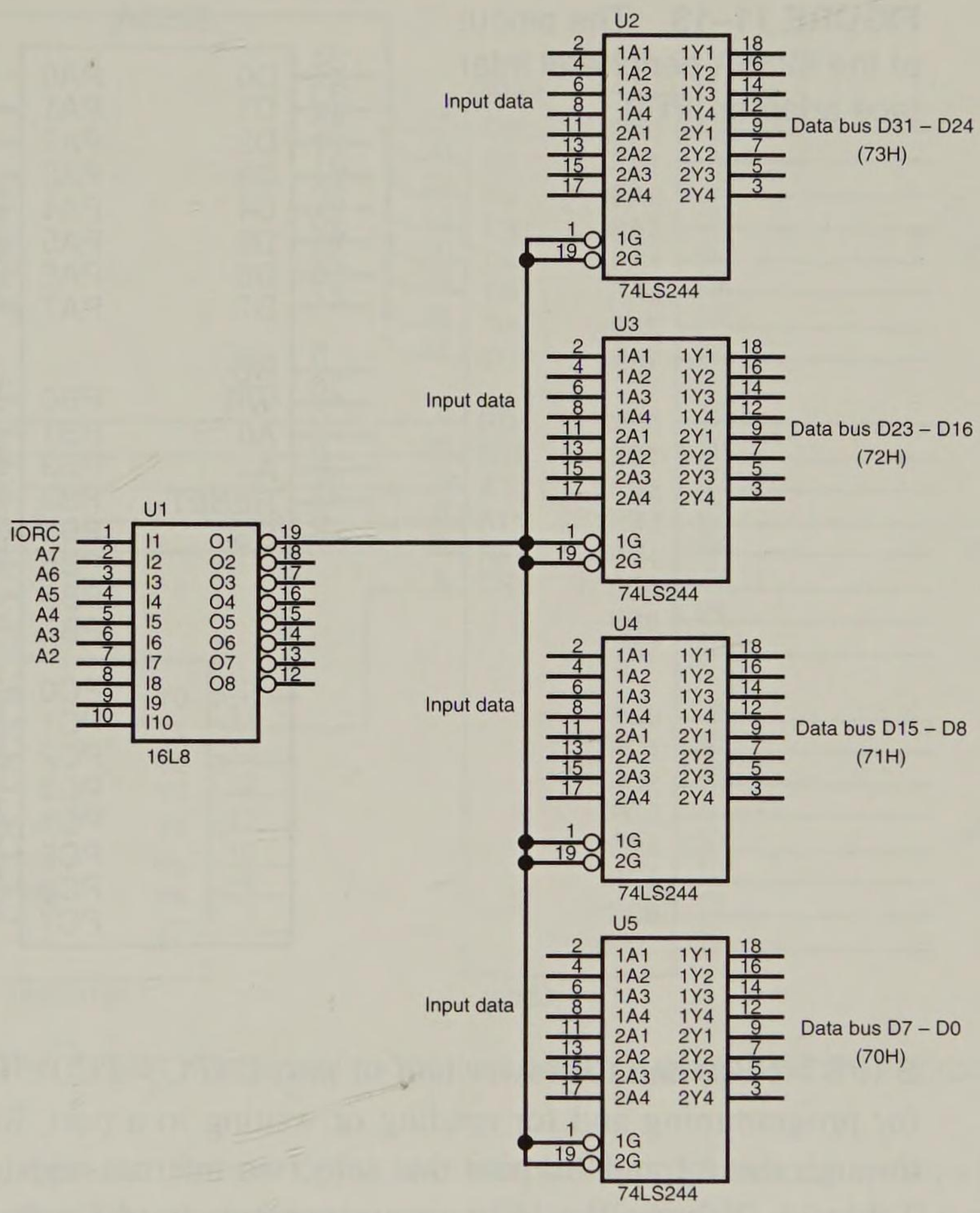



**FIGURE 11–12** A 32-bit input port decoded at bytes 70H–73H.

## 11–3 THE PROGRAMMABLE PERIPHERAL INTERFACE

The 8255 *programmable peripheral interface* (PPI) is a very popular low-cost interfacing component found in many applications. The PPI has 24 pins for I/O, programmable in groups of 12 pins, that are used in three separate modes of operation. The 8255 can interface any TTL-compatible I/O device to the microprocessor. The 82C55A (CMOS version) requires the insertion of wait states if operated with a microprocessor using higher than an 8 MHz clock. Because I/O devices are inherently slow, wait states used during I/O transfers do not impact significantly upon the speed of the system. The 8255 still finds application in the latest 80486- or Pentium-based computer systems, although it may not appear in the system as a discrete 8255. The 8255 is compatible for programming, and it is used for interface to the keyboard and the parallel printer port in 80486- or Pentium-based personal computers.

### Basic Description of the 8255

Figure 11–13 illustrates the pinout diagram of the 8255. Its three I/O ports (labeled A, B, and C) are programmed in groups of 12 pins. Group A connections consist of port A (PA7–PA0) and the upper half of port C (PC7–PC4); group B connections consist of port

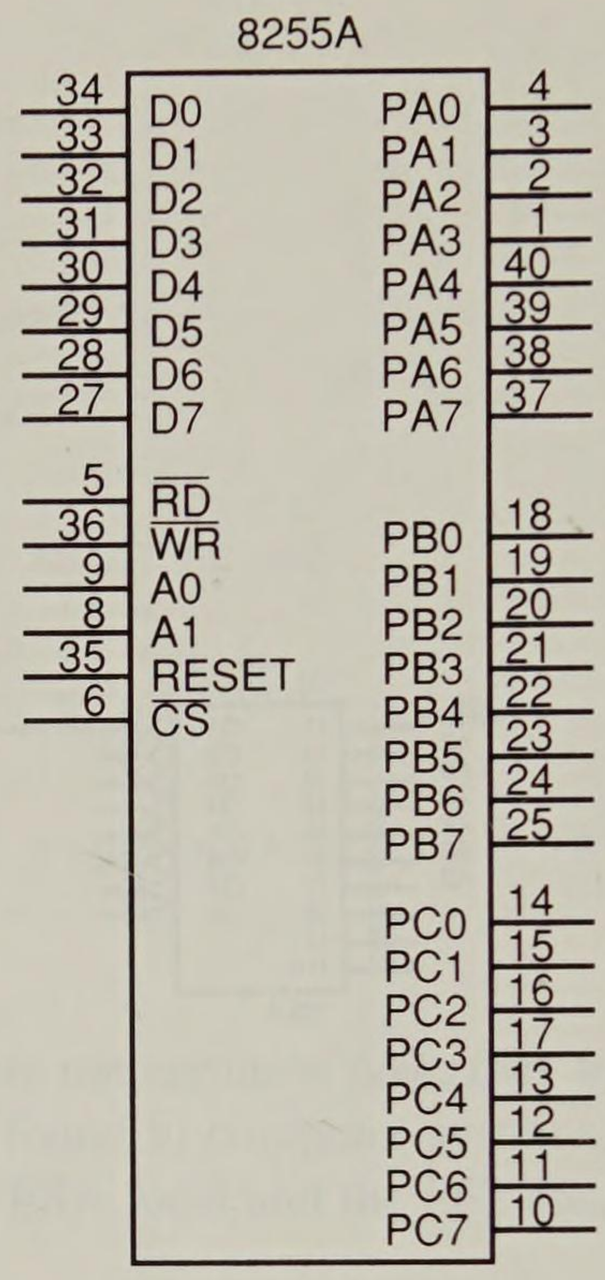



**FIGURE 11–13** The pinout of the 8255A peripheral interface adapter (PPI).

B (PB7–PB0) and the lower half of port C (PC3–PC0). The 8255 is selected by its $\overline{CS}$ pin for programming and for reading or writing to a port. Register selection is accomplished through the A1 and A0 pins that select an internal register for programming or operation. Table 11–2 shows the I/O port assignments used for programming and access to the I/O ports. In the personal computer, an 8255 or its equivalent is decoded at I/O ports 60H–63H.

The 8255 is a fairly simple device to interface to the microprocessor and program. For the 8255 to be read or written, the $\overline{CS}$ input must be a logic 0 and the correct I/O address must be applied to the A1 and A0 pins. The remaining port address pins are don't cares and are externally decoded to select the 8255.

Figure 11–14 shows an 8255 connected to the 80386SX so it functions at 8-bit I/O port addresses C0H (port A), C2H (port B), C4H (port C), and C6H (command register). This interface uses the low bank of the 8086 I/O map. Notice from this interface that all the 8255 pins are direct connections to the 80386SX, except for the $\overline{CS}$ pin. The $\overline{CS}$ pin is decoded and selected by a 74ALS138 decoder.

The RESET input to the 8255 initializes the device whenever the microprocessor is reset. A RESET input to the 8255 causes all ports to be set up as simple input ports using

**TABLE 11–2** I/O port assignments for the 8255

| *A1* | *A0* | *Function* |
|---|---|---|
| 0 | 0 | Port A |
| 0 | 1 | Port B |
| 1 | 0 | Port C |
| 1 | 1 | Command register |

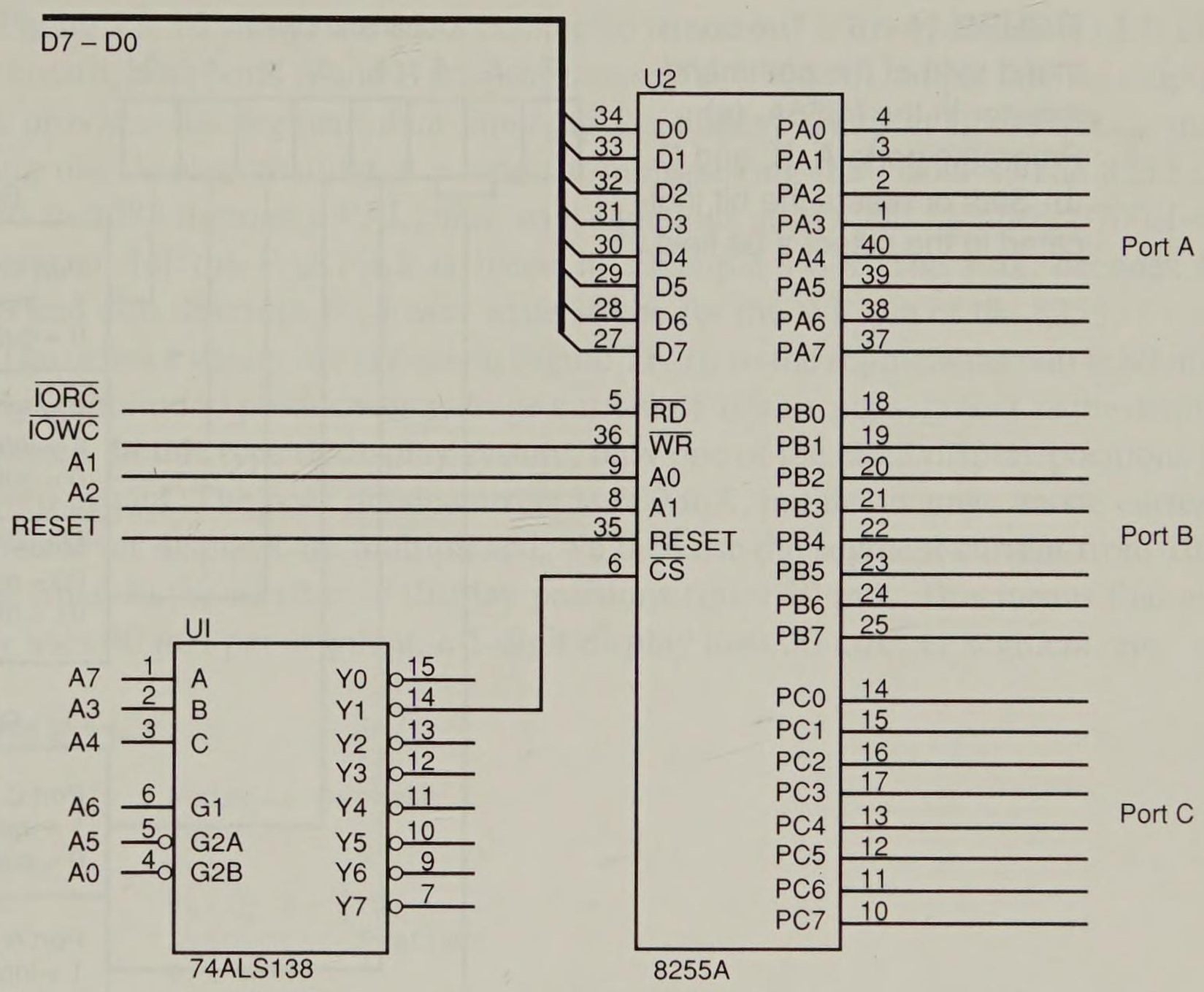



**FIGURE 11–14** The 8255A interfaced to the low bank of the 80286 microprocessor.

mode 0 operation. Because the port pins are internally programmed as input pins on a reset, this prevents damage when the power is first applied to the system. After a RESET no other commands are needed to program the 8255 as long as it used as an input device at all three ports. Note that an 8255 is interfaced to the personal computer at port addresses 60H–63H for keyboard control and also for controlling the speaker, timer, and other internal devices such as memory expansion. This is true for any AT or earlier style personal computer system.

## Programming the 8255

The 8255 is easy to program because it only contains two internal command registers as illustrated in Figure 11–15. Notice that bit position 7 selects either command byte A or command byte B. Command byte A programs the function of group A and B, while command byte B sets (1) or resets (0) bits of port C only if the 8255 is programmed in mode 1 or 2.

Group B pins (port B and the lower part of port C) are programmed as either input or output pins. Group B can operate in either mode 0 or mode 1. Mode 0 is the basic input\output mode that allows the pins of group B to be programmed as simple input and latched output connections. Mode 1 operation is the strobed operation for group B connections, where data are transferred through port B and handshaking signals are provided through port C.

Group A pins (port A and the upper part of port C) are also programmed as either input or output pins. The difference is that group A can operate in modes 0, 1, and 2. Mode 2 operation is a bidirectional mode of operation for port A.

If a 0 is placed in bit position 7 of the command byte, command byte B is selected. This command allows any bit of port C to be set (1) or reset (0) if the 8255 is operated in

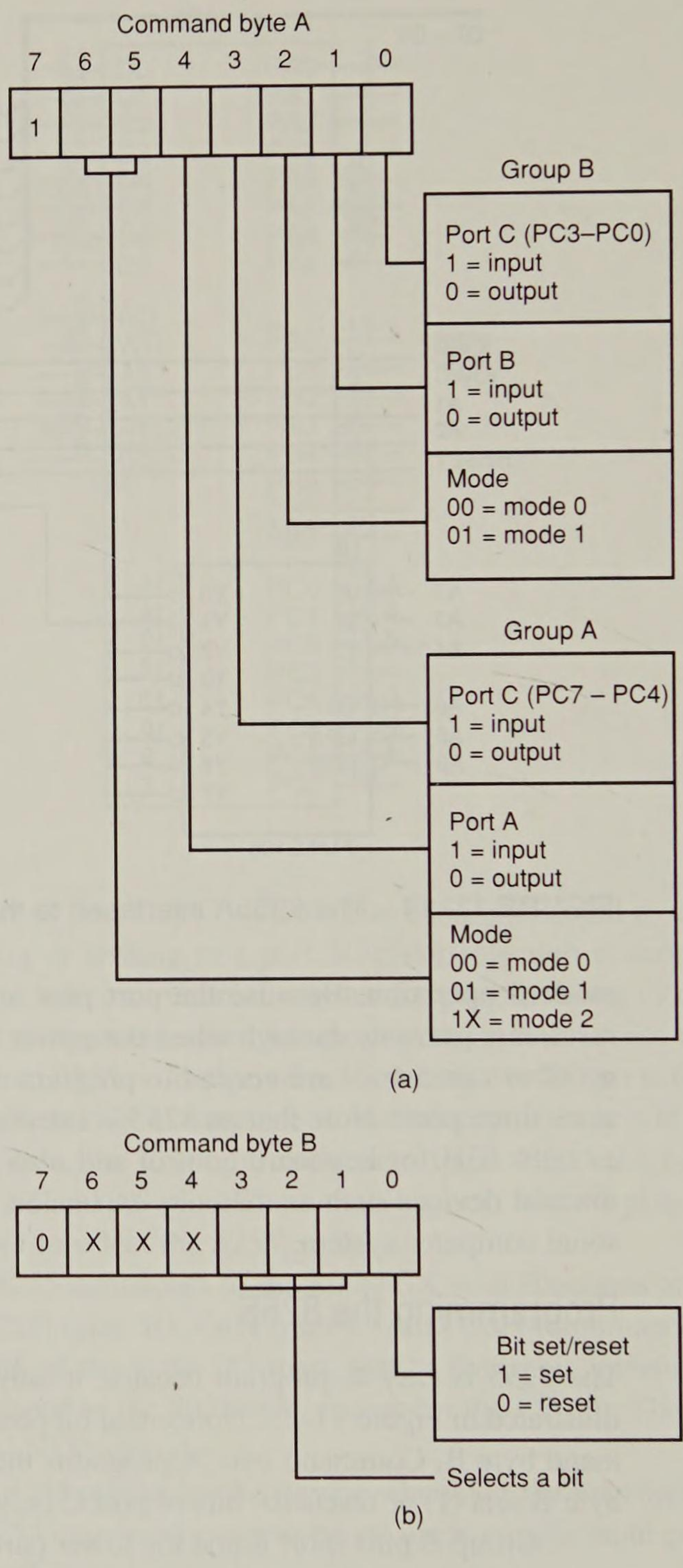



**FIGURE 11–15** The command byte of the command register in the 8255A. (a) Programs ports A, B, and C. (b) Sets or resets the bit indicated in the select a bit field.

either mode 1 or 2. Otherwise, this command byte is not used for programming. We often use the bit set/reset function in a control system to set or clear a control bit at port C.

## Mode 0 Operation

Mode 0 operation allows the 8255 to function as either a buffered input device or as a latched output device. These are the same as the basic input and output circuits discussed in the first section of this chapter.

Figure 11–16 shows the 8255 connected to a set of 8 seven-segment LED displays. In this circuit, both ports A and B are programmed as (mode 0) simple latched output ports. Port A provides the segment data inputs to the display and port B provides a means of selecting one display position at a time for multiplexing the displays. The 8255 is interfaced to an 8088 through a PAL16L8, so it functions at I/O port numbers 0700H–0703H. The program for the PAL16L8 is listed in Example 11–8. The PAL decodes the I/O address and also develops the lower write strobe for the $\overline{WR}$ pin of the 8255.

The resistor values are chosen in Figure 11–16 so the segment current is 80 mA. This current is required to produce an average current of 10 mA per segment as the displays are multiplexed. In this type of display system, only one of the eight display positions is on at any given instant. The peak anode current is 560 mA, but the average anode current is 70 mA. Whenever displays are multiplexed, we increase the segment current from 10 mA to a value equal to the number of display positions times 10 mA. This means that a 4-digit display uses 40 mA per segment, a 5-digit display uses 50 mA per segment, etc.

**EXAMPLE 11–8**

```
TITLE         Address Decoder
PATTERN       Test 8
REVISION      A
AUTHOR        Barry B. Brey
COMPANY       Symbiotic Systems
DATE          4/29/94
CHIP          Decoder8 PAL16L8

;pins     1   2   3   4   5   6   7   8   9   10
          A2  A3  A4  A5  A6  A7  A8  A9  A10 GND

;pins     11   12 13   14   15   16   17   18 19   20
          A11  CS IOM  A12  A13  A14  A15  NC NC   VCC

EQUATIONS

/CS = /A15*/A14*/A13*/A12*/A11*A10*A9*A8*/A7*/A6*/A5*/A3*A2*IOM
```

**EXAMPLE 11–9**

```
                        ;programming the 8255A
                        ;
0000 B0 80                      MOV  AL,10000000B    ;command register
0002 BA 0703                    MOV  DX,COMMAND      ;address command register
0005 EE                         OUT  DX,AL
```

Before software to operate the display is examined, we must program the 8255. This is accomplished with the short sequence of instructions listed in Example 11–9. Here port A and B are both programmed as outputs.

**EXAMPLE 11–10**

```
                        ;Procedure to scan the 8 digits of the multiplexed display.
                        ;This procedure must be called continuously from a program
                        ;to display 7-segment coded information from memory.
                        ;
0006                    DISP   PROC NEAR
```

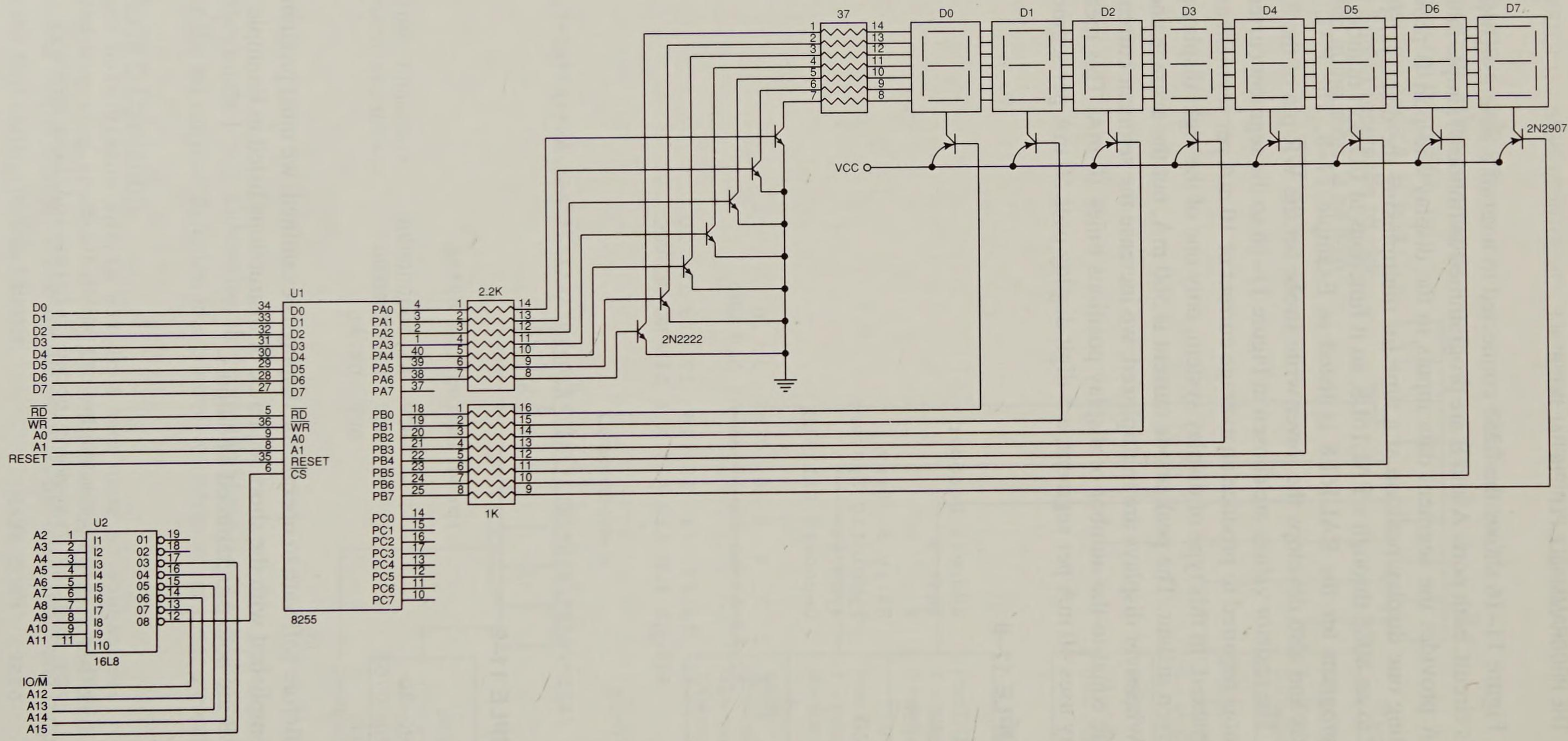



**FIGURE 11–16** An 8-digit LED display interfaced to the 8088 microprocessor through an 8255A PIA.

```
0006 9C                 PUSHF                   ;save registers
0007 50                 PUSH AX
0008 53                 PUSH BX
0009 52                 PUSH DX
000A 56                 PUSH SI

                ;set up register for display

000B BB 0008            MOV  BX,8               ;load count
000E B4 7F              MOV  AH,7FH             ;load select pattern
0010 BE FFFF R          MOV  SI,OFFSET MEM-1    ;address display RAM
0013 BA 0701            MOV  DX,PORTB           ;address port B

                ;display 8 digits

0016                    DISP1:
0016 8A C4                   MOV  AL,AH         ;select digit
0018 EE                      OUT  DX,AL
0019 4A                      DEC  DX            ;address port A
001A 8A 00                   MOV  AL,[BX+SI]    ;get data
001C EE                      OUT  DX,AL         ;display data
001D E8 029A R               CALL DELAY         ;wait 1 ms
0020 D0 CC                   ROR  AH,1          ;adjust select code
0022 42                      INC  DX            ;address port B
0023 4B                      DEC  BX            ;adjust count
0024 75 F0                   JNZ  DISP1         ;repeat 8 times

0026 5E                 POP  SI                 ;restore registers
0027 5A                 POP  DX
0028 5B                 POP  BX
0029 58                 POP  AX
002A 9D                 POPF
002B C3                 RET

002C                DISP    ENDP
```

The procedure to drive these displays is listed in Example 11–10. For this display system to function correctly, we must call this procedure often. Notice that this procedure calls another procedure (DELAY), which causes a 1 ms time delay. This time delay is not illustrated in this example, but is used to allow time for each display position to turn on. It is recommended by the manufacturers of LED displays that the display flash is between 100 Hz and 1,500 Hz. Using a 1 ms time delay, we light each digit for 1 ms, for a total display flash rate of 1,000 Hz / 8 display or a flash rate of 125.

The display procedure addresses an area of memory where the data, in 7-segment code, is stored for the 8 display digits. The AH register is loaded with a code (7FH) that initially addresses the most significant display position. Once this position is selected, the content of memory location MEMORY +7 is addressed and sent to the most significant digit. The selection code is then adjusted to select the next display digit as is the address. This process repeats 8 times to display the contents of location MEMORY through MEMORY +7 on the 8 display digits.

*A Stepper Motor Interfaced to the 8255.* Another device often interfaced to a computer system is the stepper motor. A stepper motor is a digital motor moved in discrete steps as it traverses through 360°. Stepper motors are commonly geared to move 15° per step (in an inexpensive stepper motor) to 1° per step (on a more costly, high-precision stepper

motor). In all cases, these steps are gained through many magnetic poles and/or gearing. Notice that two coils are energized in Figure 11–17. If less power is required, one coil may be energized at a time, causing the motor to step at 0°, 90°, 180°, and 270°.

Figure 11–17 shows a four-coil stepper motor that uses an armature with a single pole. In this illustration, the stepper motor is shown four times, with the permanent magnetic armature rotated to four discrete places. This is accomplished by energizing the coils as shown. This is an illustration of full stepping. The stepper motor is driven using NPN Darlington pairs to provide a large current to each coil.

A circuit that can drive this stepper motor is illustrated in Figure 11–18, with the four coils shown in place. This circuit uses the 8255 to provide it with the drive signals used to rotate the armature of the motor in either a right-hand or left-hand direction.

**EXAMPLE 11–11**

```
0000                    DATA    SEGMENT

0000 33                 POS             DB      33H             ;position

0001                    DATA    ENDS

0000                    CODE    SEGMENT  'CODE'
                                ASSUME  CS:CODE,DS:DATA
```

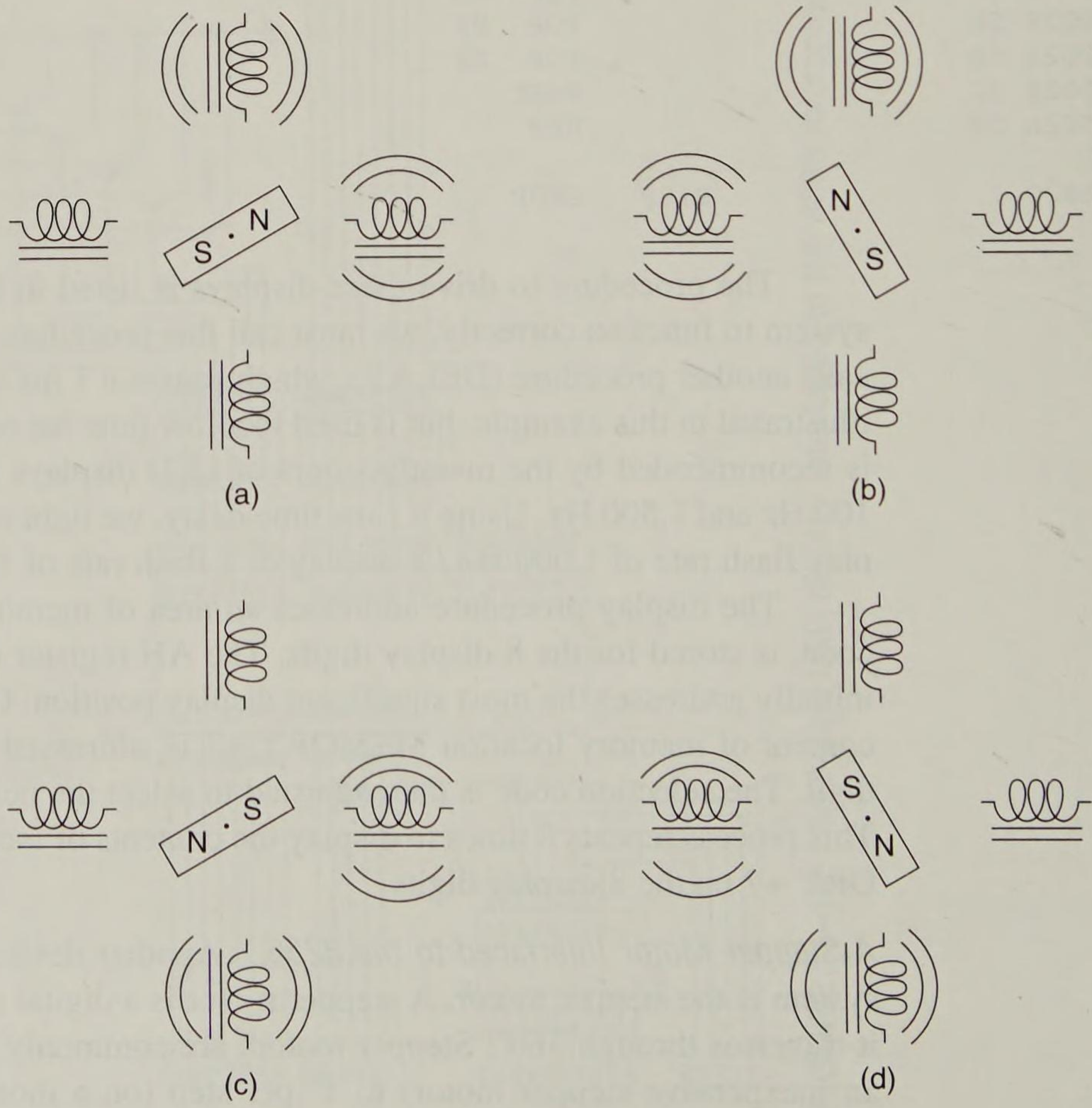



**FIGURE 11–17** The stepper motor showing full-step operation. (a) 45°, (b) 135°, (c) 225°, (d) 317°.

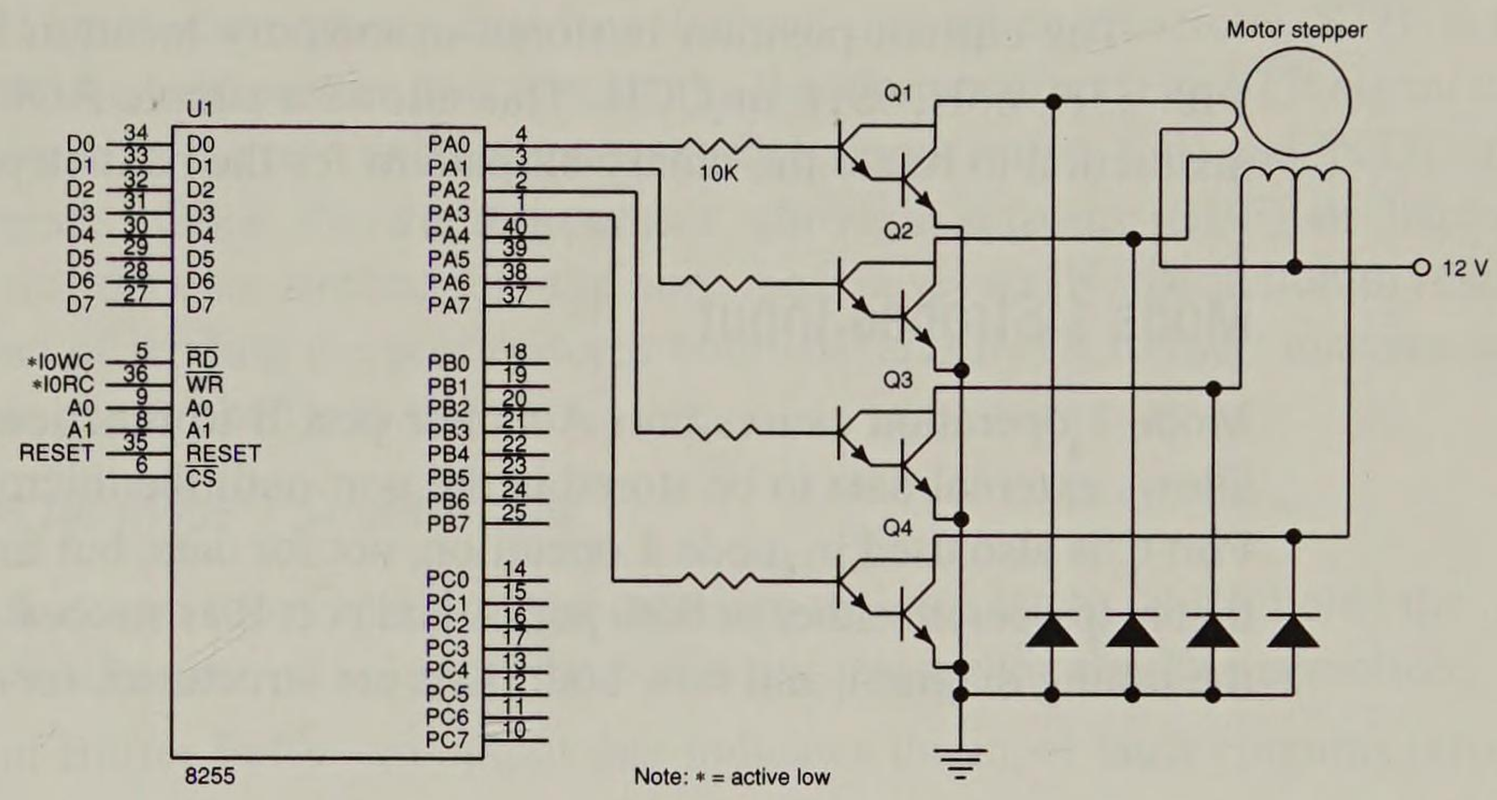



**FIGURE 11–18** A stepper motor interfaced to the 8255. This illustration does not show the decoder.

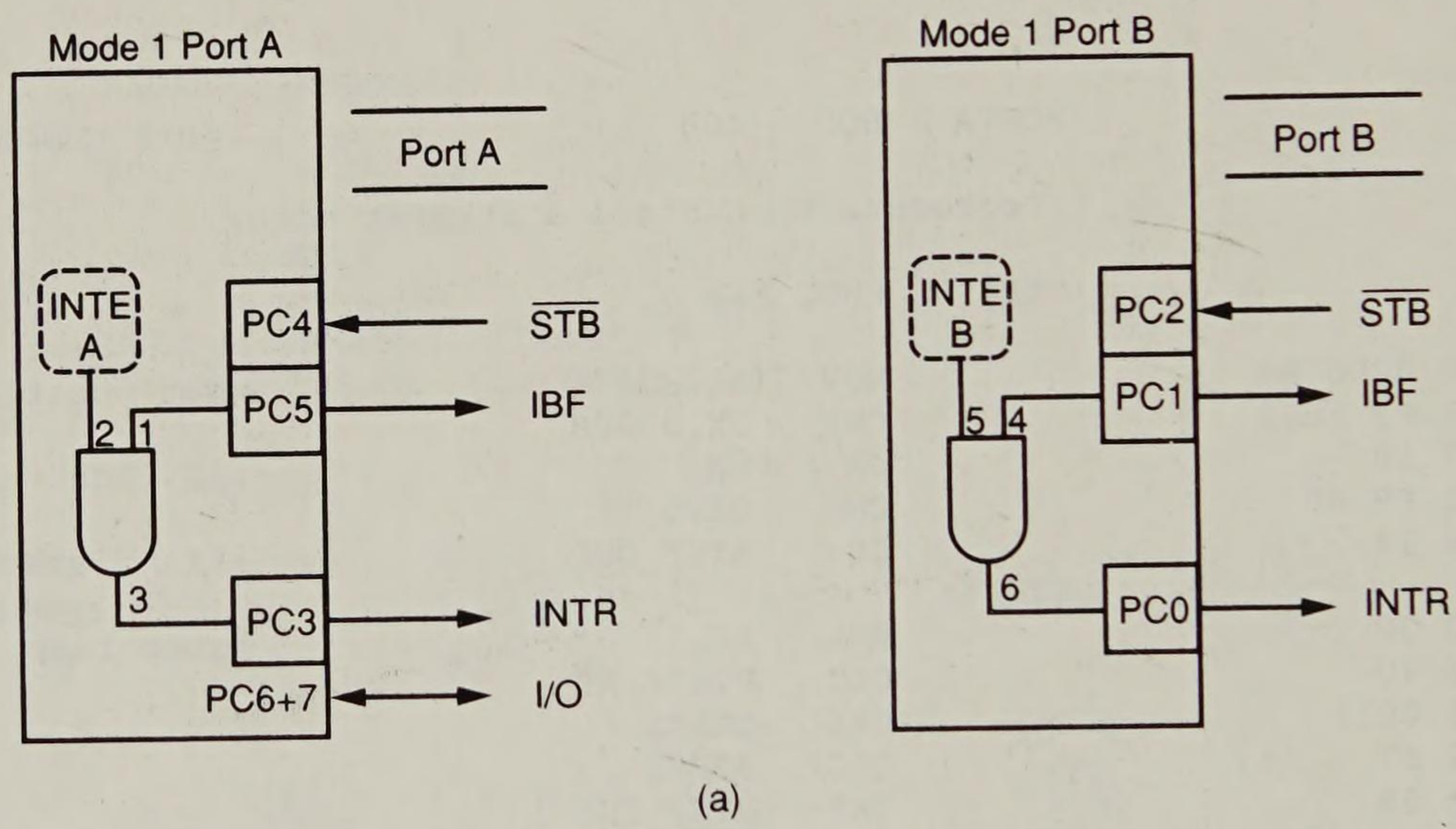
```
= 0040                  PORTA   EQU   40H                     ;port number

                        ;Procedure to control a stepper motor.

0000                    STEP    PROC FAR

0000 A0 0000 R                  MOV   AL,POS                  ;get position
0003 81 F9 8000                 CMP   CX,8000H
0007 77 10                      JA    RH                      ;if right-hand direction
0009 83 F9 00                   CMP   CX,0
000C 74 14                      JE    STEP_OUT                ;if no steps
000E                    STEP1:
000E D0 C0                      ROL   AL,1                    ;step left
0010 E6 40                      OUT   PORTA,AL
0012 E8 0011                    CALL  DELAY
0015 E2 F7                      LOOP  STEP1
0017 EB 09                      JMP   STEP_OUT
0019                    RH:
0019 81 E1 7FFF                 AND   CX,7FFFH                ;clear bit 15
001D                    RH1:
001D D0 C8                      ROR   AL,1                    ;step right
001F E6 40                      OUT   PORTA,AL
0021 E8 0006                    CALL  DELAY
0024 E2 F7                      LOOP  RH1
0026                    STEP_OUT:
0026 A2 0000                    MOV   POS,AL                  ;save position
0029 C9                         RET

002A                    STEP    ENDP
002A                    CODE    ENDS
                                END   STEP
```

A simple procedure that drives the motor (assuming port A is programmed in Mode 0 as an output device) is listed in Example 11–11. This subroutine is called with CX holding the number of steps and direction of the rotation. If CX > 8000H, the motor spins in the right-hand direction; if CX < 8000H it spins in the left-hand direction. The left-most bit of CX is removed and the remaining 15 bits contain the number of steps. Notice that the procedure uses a time delay (not illustrated) that causes a 1 ms time delay. This time delay allows the stepper motor armature time to move to its next position.

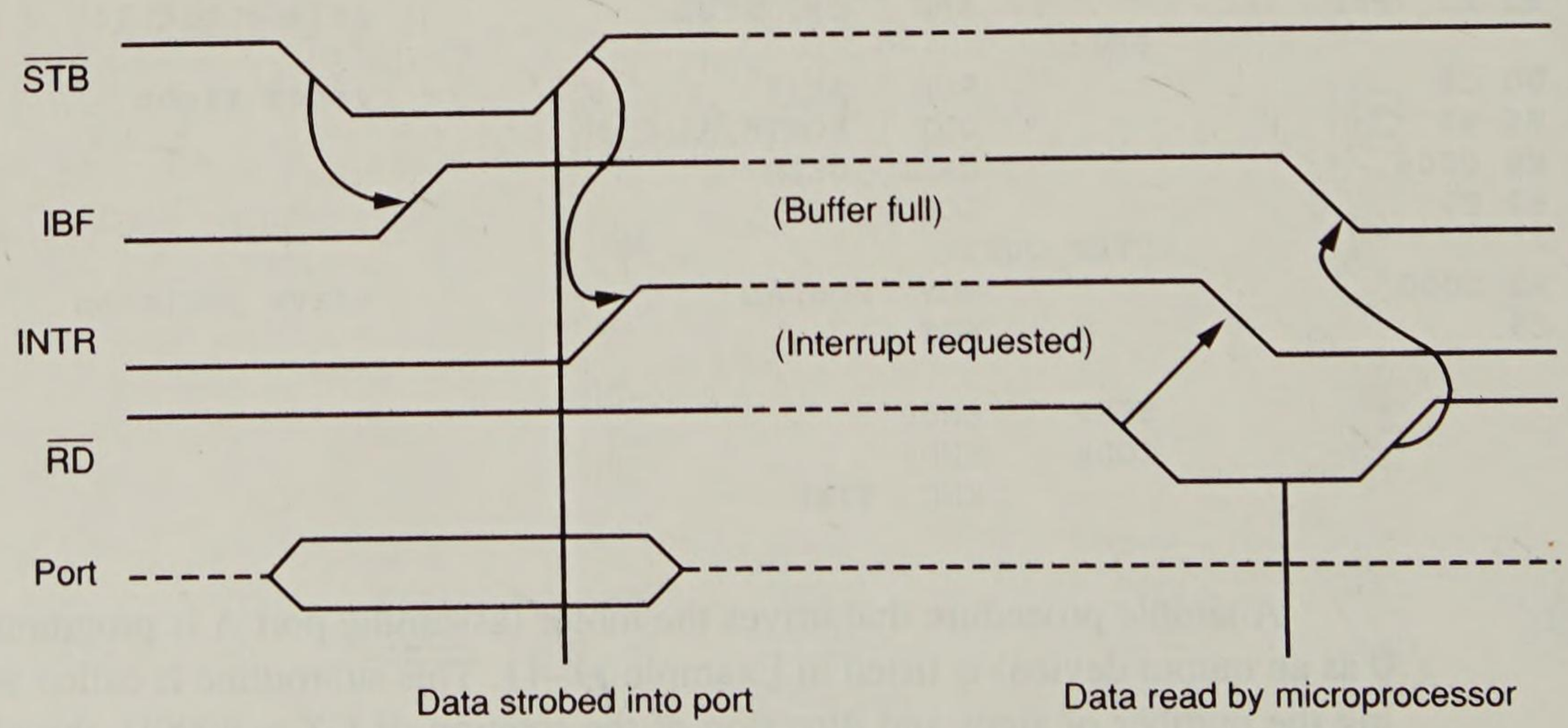
The current position is stored in memory location POS, which must be initialized with 33H, 66H, 99H, or CCH. This allows a simple ROR (step right) or ROL (step left) instruction to rotate the binary bit pattern for the next step.

## Mode 1 Strobed Input

Mode 1 operation causes port A and/or port B to function as latching input devices. This allows external data to be stored in the port until the microprocessor is ready to retrieve it. Port C is also used in mode 1 operation, not for data, but for control or handshaking signals that help operate either or both port A and port B as strobed input ports. Figure 11–19 shows the timing diagram, and how both ports are structured, for mode 1 strobed input operation.




(a)




(b)

**FIGURE 11–19** Strobed input operation (mode 1) of the 8255A. (a) Internal structure. (b) Timing diagram.

The strobed input port captures data from the port pins when the strobe ($\overline{STB}$) is activated. Note that strobe captures the port data on the 0-to-1 transition. The $\overline{STB}$ signal causes data to be captured in the port and activates the IBF (input buffer full) and INTR (interrupt request) signals. Once the microprocessor, through software (IBF) or hardware (INTR), notices that data are strobed into the port, it executes an IN instruction to read the port ($\overline{RD}$). The act of reading the port restores both IBF and INTR to their inactive states until the next data are strobed into the port.

### *Signal Definitions for Mode 1 Strobed Input*

| | |
|---|---|
| $\overline{STB}$ | (Strobe)—an input used to load data into the port latch, which holds the information until it is input to the microprocessor via the IN instruction. |
| $\overline{IBF}$ | (Input Buffer Full)—an output that indicates the input latch contains information. |
| INTR | (Interrupt Request)—an output that requests an interrupt. The INTR pin becomes a logic 1 when the $\overline{STB}$ input returns to a logic 1, and is cleared when the data are input from the port by the microprocessor. |
| INTE | (Interrupt Enable)—neither an input nor an output, but an internal bit programmed via the port PC4 (port A) or PC2 (port B) bit position. |
| PC7, PC6 | (Port Pins 7 and 6)—general purpose I/O pins that are available for any purpose. |

*Strobe Input Example.* An excellent example of a strobed input device is a keyboard. The keyboard encoder debounces the keyswitches and provides a strobe signal whenever a key is depressed, and the data output contains the ASCII-coded keycode. Figure 11–20 illustrates a keyboard connected to strobed input port A. Here $\overline{DAV}$ (data available) is activated for 1 μs each time a key is typed on the keyboard. This causes data to be strobed into port A, because $\overline{DAV}$ is connected to the $\overline{STB}$ input of port A. So each time a key is typed, it is stored into port A of the 8255. The $\overline{STB}$ input also activates the IBF signal, indicating that data is in port A.

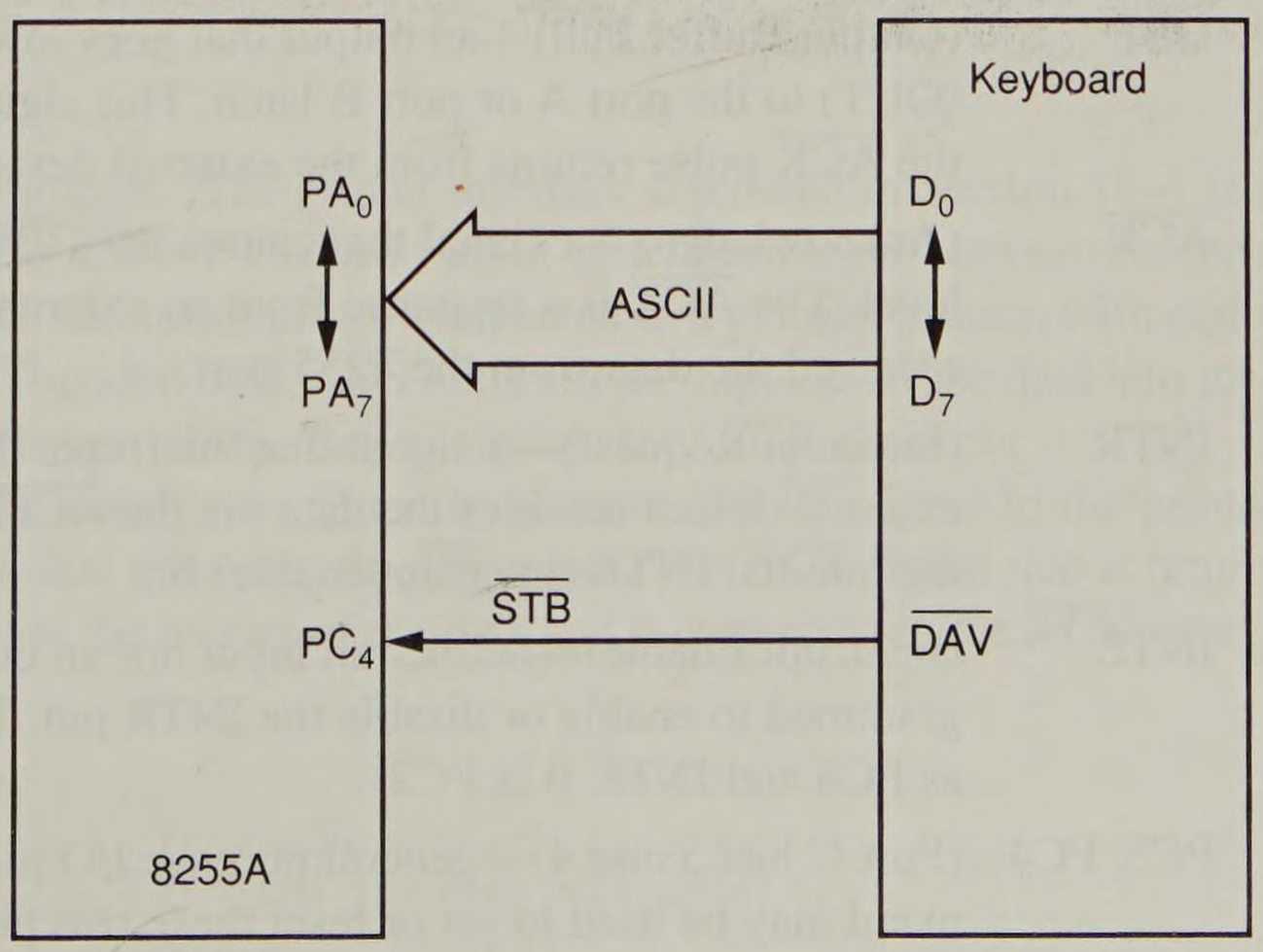



**FIGURE 11–20** Using the 8255A for strobed input operation of a keyboard.

Example 11–12 shows a procedure that reads data from the keyboard each time a key is typed. This procedure reads the key from port A and returns with the ASCII code in AL. To detect a key, port C is read and the IBF bit (bit position PC5) is tested to see if the buffer is full. If the buffer is empty (IBF = 0), then the procedure keeps testing this bit, waiting for a character to be typed on the keyboard.

**EXAMPLE 11–12**

```
                        ;Procedure that reads the keyboard encoder and returns
                        ;with the ASCII character in AL.
                        ;
= 0020                  BIT5    EQU  20H

0000                    READ    PROC NEAR
0000 E4 22                      IN   AL,PORTC        ;read port C
0002 A8 20                      TEST AL,BIT5         ;test IBF
0004 74 FA                      JZ   READ            ;if IBF = 0

0006 E4 20                      IN   AL,PORTA        ;read ASCII data
0008 C3                         RET

0009                    READ    ENDP
```

## MODE 1 STROBED OUTPUT

Figure 11–21 illustrates the internal configuration and timing diagram of the 8255 when it is operated as a strobed output device under mode 1. Strobed output operation is similar to mode 0 output, except that control signals are included to provide handshaking.

Whenever data are written to a port programmed as a strobed output port, the $\overline{\text{OBF}}$ (output buffer full) signal becomes a logic 0, to indicate that data are present in the port latch. This signal indicates that data are available to an external I/O device that removes the data by strobing the $\overline{\text{ACK}}$ (acknowledge) input to the port. The $\overline{\text{ACK}}$ signal returns the $\overline{\text{OBF}}$ signal to a logic 1, indicating that the buffer is not full.

### *Signal Definitions for Mode 1 Strobed Output*

$\overline{\text{OBF}}$ (Output Buffer Full)—an output that goes low whenever data are output (OUT) to the port A or port B latch. This signal is set to a logic 1 whenever the $\overline{\text{ACK}}$ pulse returns from the external device.

$\overline{\text{ACK}}$ (Acknowledge)—a signal that causes the $\overline{\text{OBF}}$ pin to return to a logic 1 level. The $\overline{\text{ACK}}$ is a response from an external device that indicates it has received the data from the 8255 port.

INTR (Interrupt Request)—a signal that interrupts the microprocessor when the external device receives the data via the $\overline{\text{ACK}}$ signal. This pin is qualified by the internal INTE (interrupt enable) bit.

INTE (Interrupt Enable)—neither an input nor an output, but an internal bit programmed to enable or disable the INTR pin. The INTE A bit is programmed as PC6 and INTE B is PC2.

PC5, PC4 (Port C bits 5 and 4)—general purpose I/O pins. The bit set and reset command may be used to set or reset these two pins.

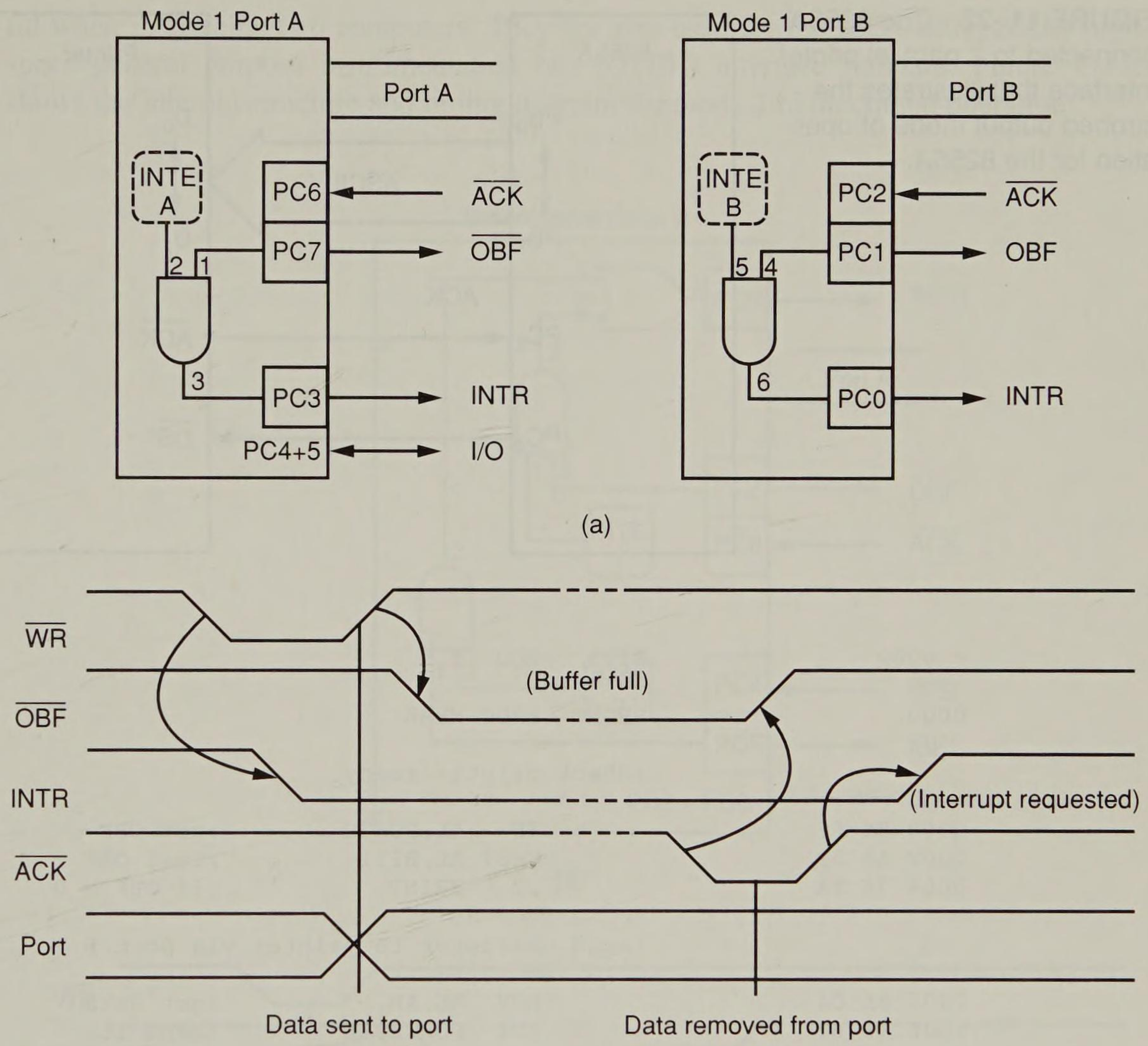



**FIGURE 11–21** Strobed output operation (mode 1) of the 8266A. (a) Internal structure. (b) Timing diagram.

*Strobed Output Example.* The printer interface discussed in Section 11–1 is used here to demonstrate how to achieve strobed output synchronization between the printer and the 8255. Figure 11–22 illustrates port B connected to a parallel printer, with eight data inputs for receiving ASCII-coded data, a $\overline{DS}$ (data strobe) input to strobe data into the printer, and an $\overline{ACK}$ output to acknowledge the receipt of the ASCII character.

In this circuit, there is no signal to generate the $\overline{DS}$ signal to the printer, so PC4 is used with software that generates the $\overline{DS}$ signal. The $\overline{ACK}$ signal that is returned from the printer acknowledges the receipt of the data and is connected to the $\overline{ACK}$ input of the 8255.

**EXAMPLE 11–13**

```
;Procedure that transfers the ASCII character from AH
;to the printer via port B.
;
```

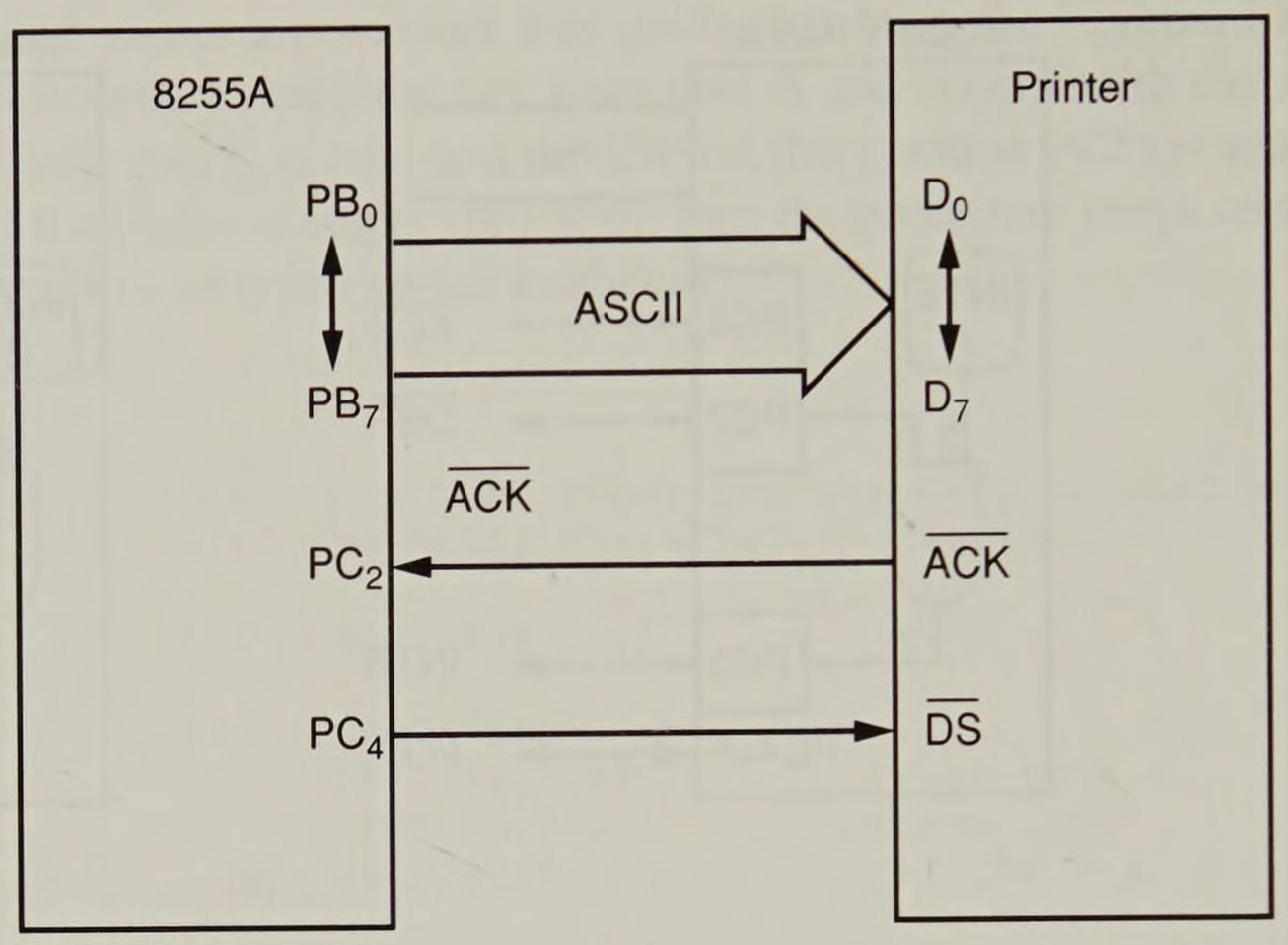



**FIGURE 11–22** The 8255A connected to a parallel printer interface that illustrates the strobed output mode of operation for the 8255A.

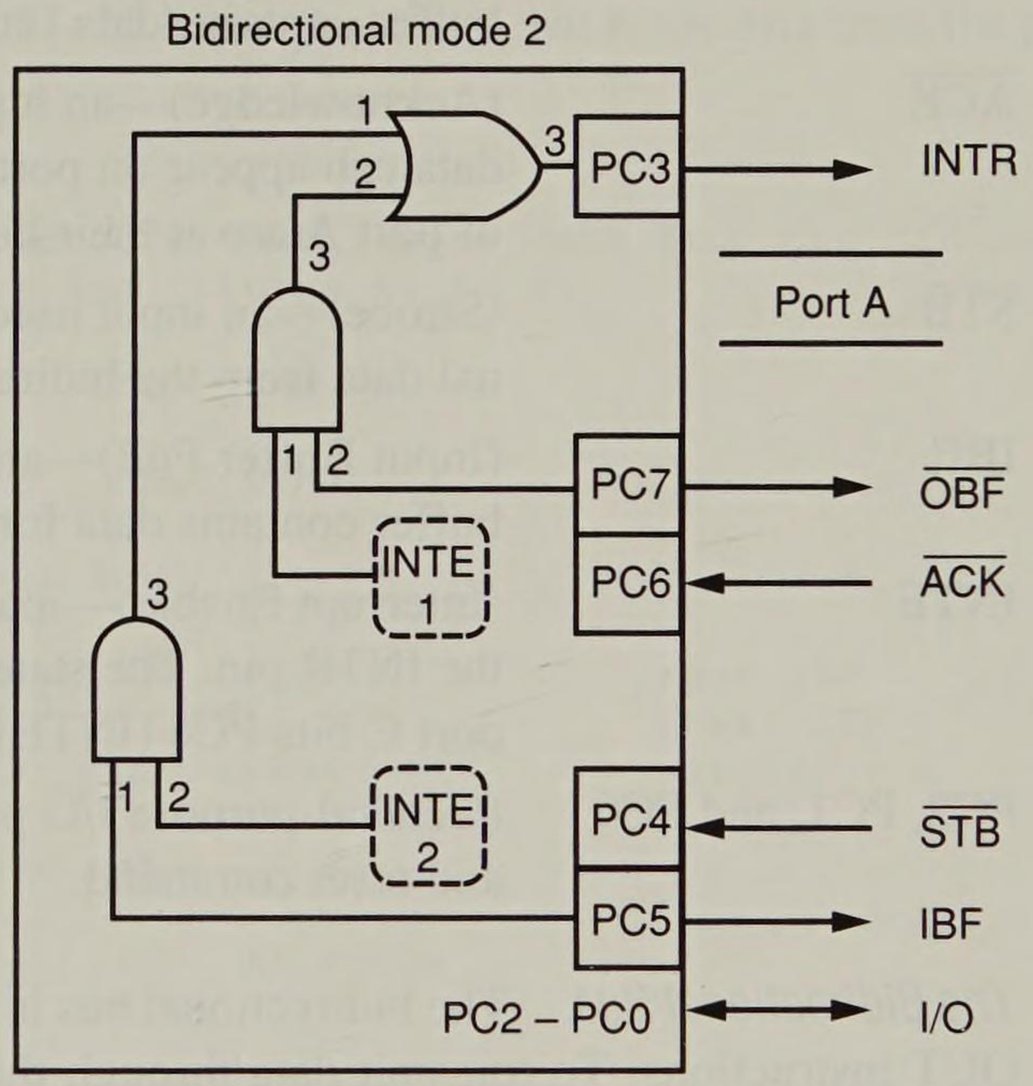
```
= 0002                  BIT1    EQU  2
0000                    PRINT   PROC NEAR
                        ;check printer ready
0000 E4 62                      IN   AL,PORTC          ;get OBF
0002 A8 02                      TEST AL,BIT1           ;test OBF
0004 74 FA                      JZ   PRINT             ;if OBF = 0
                      ;send character to printer via port B
0006 8A C4                      MOV  AL,AH             ;get data
0008 E6 61                      OUT  PORTB,AL          ;send it
                      ;send DS to printer
000A B0 08                      MOV  AL,8              ;clear DS
000C E6 63                      OUT  COMMAND,AL
000E B0 09                      MOV  AL,9              ;set DS
0010 E6 63                      OUT  COMMAND,AL
0012 C3                         RET
0013                    PRINT   ENDP
```

Example 11–13 lists the software that sends the ASCII-coded character in AH to the printer. The procedure first tests $\overline{\text{OBF}}$ to decide whether the printer has removed the data from port B. If not, the procedure waits for the $\overline{\text{ACK}}$ signal to return from the printer. If $\overline{\text{OBF}} = 1$, then the procedure sends the contents of AH to the printer through port B, and also sends the $\overline{\text{DS}}$ signal.

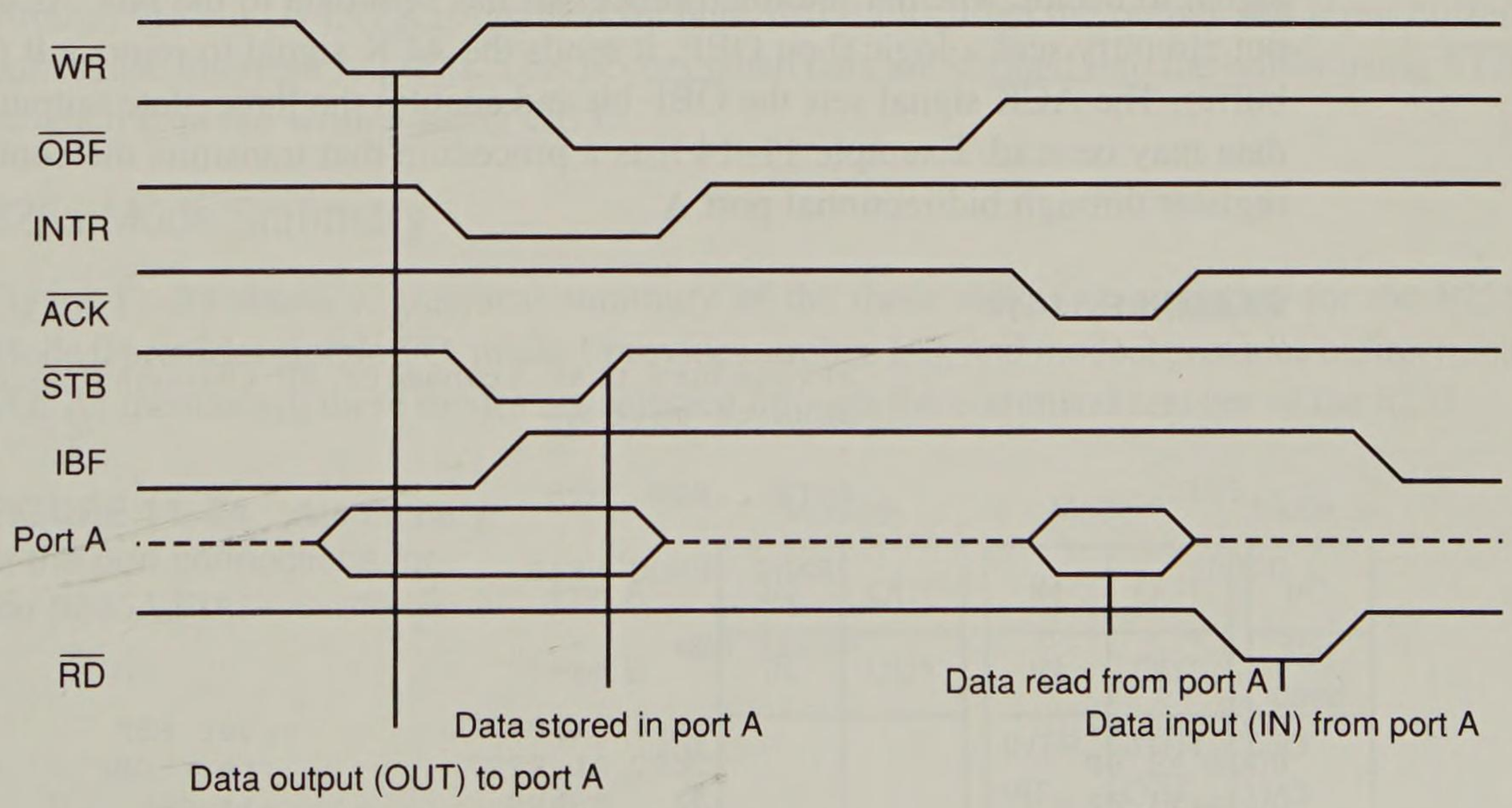
## Mode 2 Bidirectional Operation

In mode 2, which is allowed only for group A, port A becomes bidirectional, allowing data to be transmitted and received over the same eight wires. Bidirectional bused data are use-

ful when interfacing two computers. They are also used for the IEEE–488 parallel high-speed general purpose instrumentation bus (GPIB*) interface standard. Figure 11–23 shows the internal structure and timing diagram for mode 2 bidirectional operation.




(a)




(b)

**FIGURE 11–23** Mode 2 operation of the 8255A. (a) Internal structure. (b) Timing diagram.

*GPIB (general purpose instrumentation bus) is a registered trademark of Hewlett-Packard Corporation.

*Signal Definitions for Bidirectional Mode 2*

| | |
|---|---|
| INTR | (Interrupt Request)—an output used to interrupt the microprocessor for both input and output conditions. |
| $\overline{\text{OBF}}$ | (Output Buffer Full)—an output that indicates that the output buffer contains data for the bidirectional bus. |
| $\overline{\text{ACK}}$ | (Acknowledge)—an input that enables the three-state buffers so data can appear on port A. If $\overline{\text{ACK}}$ is a logic 1, the output buffers of port A are at their high-impedance state. |
| $\overline{\text{STB}}$ | (Strobe)—an input used to load the port A input latch with external data from the bidirectional port A bus. |
| IBF | (Input Buffer Full)—an output used to signal that the input buffer contains data for the external bidirectional bus. |
| INTE | (Interrupt Enable)—internal bits (INTE1 and INTE2) that enable the INTR pin. The state of the INTR pin is controlled through port C bits PC6 (INTE1) and PC4 (INTE2). |
| PC2, PC1, and PC0 | (General-purpose I/O pins in mode 2)—controlled by the bit set and reset command. |

*The Bidirectional Bus.* The bidirectional bus is used by referencing port A with the IN and OUT instructions. To transmit data through the bidirectional bus, the program first tests the $\overline{\text{OBF}}$ signal to determine whether the output buffer is empty. If it is, then data are sent to the output buffer via the OUT instruction. The external circuitry also monitors the $\overline{\text{OBF}}$ signal to decide whether the microprocessor has sent data to the bus. As soon as the output circuitry sees a logic 0 on $\overline{\text{OBF}}$, it sends the $\overline{\text{ACK}}$ signal to remove it from the output buffer. The $\overline{\text{ACK}}$ signal sets the $\overline{\text{OBF}}$ bit and enables the three-state output buffers so that data may be read. Example 11–14 lists a procedure that transmits the contents of the AH register through bidirectional port A.

**EXAMPLE 11–14**

```
                        ;Procedure that transmits AH through the bidirectional
                        ;bus of port A
                        ;
= 0080                  BIT7    EQU   80H

0000                    TRANS   PROC NEAR

                        ;test OBF

0000 E4 62                      IN    AL,PORTC          ;get OBF
0002 A8 80                      TEST  AL,BIT7           ;test OBF
0004 74 FA                      JZ    TRANS             ;if OBF = 0

                        ;send data

0006 8A C4                      MOV   AL,AH             ;get data
0008 E6 60                      OUT   PORTA,AL
000A C3                         RET

000B                    TRANS   ENDP
```

To receive data through the bidirectional port A bus, the IBF bit is tested with software, to decide whether data have been strobed into the port. If IBF = 1, then data are input using the IN instruction. The external interface sends data into the port using the $\overline{\text{STB}}$ signal. When $\overline{\text{STB}}$ is activated, the IBF signal becomes a logic 1, and the data at port A are held inside the port in a latch. When the IN instruction executes, the IBF bit is cleared and the data in the port are moved into AL. Example 11–15 lists a procedure that reads data from the port.

**EXAMPLE 11–15**

```
                        ;Procedure that inputs data from the bidirectional
                        ;bus and returns it in AL
                        ;
= 0020                  BIT5    EQU  20H

0000                    READ    PROC NEAR

                        ;test IBF

0000 E4 62                      IN   AL,PORTC       ;get IBF
0002 A8 20                      TEST AL,BIT5        ;test IBF
0004 74 FA                      JZ   READ           ;if IBF = 0

                        ;read data

0006 E4 60                      IN   AL,PORTA
0008 C3                         RET

0009                    READ    ENDP
```

The INTR (interrupt request) pin can be activated from data flow in either direction through the bus. If INTR is enabled by both INTE bits, then the output and input buffers both cause interrupt requests. This occurs when data are strobed into the buffer using $\overline{\text{STB}}$, or when data are written using OUT.

## 8255 Mode Summary

Figure 11–24 shows a graphical summary of the three modes of operation for the 8255. Mode 0 provides simple I/O, mode 1 provides strobed I/O, and mode 2 provides bidirectional I/O. As mentioned, these modes are selected through the command register of the 8255.

**FIGURE 11–24** A summary of the port connections for the 8255A PIA.

| | | Mode 0 | | Mode 1 | | Mode 2 |
|---|---|---|---|---|---|---|
| Port A | | IN | OUT | IN | OUT | I/O |
| Port B | | IN | OUT | IN | OUT | Not used |
| Port C | 0 | IN | OUT | $INTR_B$ | $INTR_B$ | I/O |
| | 1 | | | $IBF_B$ | $\overline{OBF}_B$ | I/O |
| | 2 | | | $\overline{STB}_B$ | $\overline{ACK}_B$ | I/O |
| | 3 | | | $INTR_A$ | $INTR_A$ | INTR |
| | 4 | | | $\overline{STB}_A$ | I/O | $\overline{STB}$ |
| | 5 | | | $IBF_A$ | I/O | IBF |
| | 6 | | | I/O | $\overline{ACK}_A$ | $\overline{ACK}$ |
| | 7 | | | I/O | $\overline{OBF}_A$ | $\overline{OBF}$ |

## 11–4 THE 8279 PROGRAMMABLE KEYBOARD/DISPLAY INTERFACE

The 8279 is a programmable keyboard and display interfacing component that scans and encodes up to a 64-key keyboard and controls up to a 16-digit numerical display. The keyboard interface has a built-in first-in, first-out (FIFO) buffer that allows it to store up to eight keystrokes before the microprocessor must retrieve a character. The display section controls up to 16 numeric displays from an internal 16 × 8 RAM that stores the coded display information.

### Basic Description of the 8279

The 8279 is designed for ease of interfacing with any microprocessor. Figure 11–25 illustrates the pinout of this device. The definition of each pin connection follows.

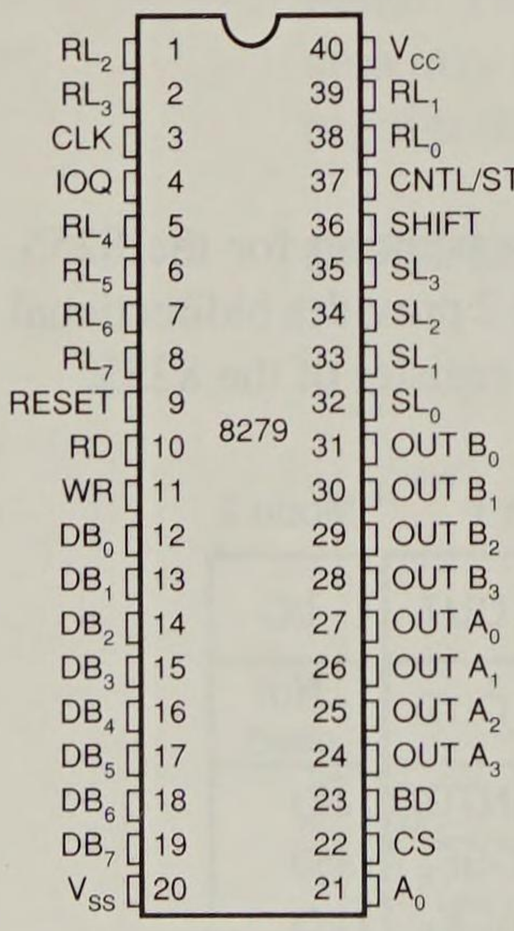
#### *Pin Definition for the 8279*

A0 (Address Input)—a pin that selects data or control for reads and writes between the microprocessor and the 8279. A logic 0 selects data and a logic 1 selects control or status register.

$\overline{BD}$ (Blank)—an output used to blank the displays.

CLK (Clock)—an input that generates the internal timing for the 8279. The maximum allowable frequency on the CLK pin is 3.125 MHz. Other timings require wait states in microprocessors executing at above 5 MHz.

PIN CONFIGURATION




PIN NAMES

| NAME | I/O | FUNCTION |
|---|---|---|
| $DB_{07}$ | I/O | DATA BUS (BIDIRECTIONAL) |
| CLK | I | CLOCK INPUT |
| RESET | I | RESET INPUT |
| CS | I | CHIP SELECT |
| RD | I | READ INPUT |
| WR | I | WRITE INPUT |
| $A_0$ | I | BUFFER ADDRESS |
| IRO | O | INTERRUPT REQUEST OUTPUT |
| $SL_{03}$ | O | SCAN LINES |
| $RL_{07}$ | I | RETURN LINES |
| SHIFT | I | SHIFT INPUT |
| CNTL/STB | I | CONTROL/STROBE INPUT |
| $OUT A_{03}$ | O | DISPLAY (A) OUTPUTS |
| $OUT B_{03}$ | O | DISPLAY (B) OUTPUTS |
| $\overline{BD}$ | O | BLANK DISPLAY OUTPUT |

LOGIC SYMBOL

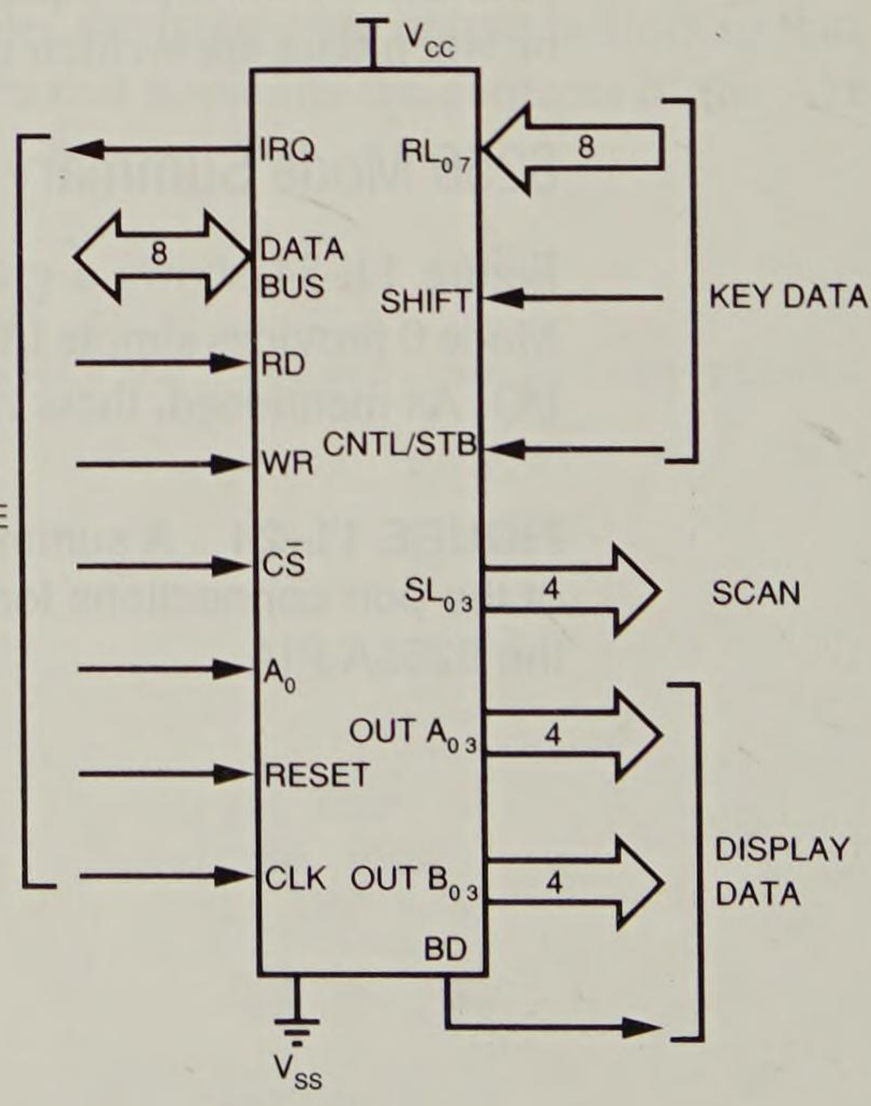



**FIGURE 11–25** The pinout and logic symbol of the 8279 programmable keyboard/display interface. (Courtesy of Intel Corporation)

| | |
|---|---|
| CN/ST | (Control/Strobe)—an input normally connected to the control key on a keyboard. |
| $\overline{CS}$ | (Chip Select)—an input used to enable the 8279 for programming, reading the keyboard and status information, and writing control and display data. |
| DB7–DB0 | (Data Bus)—bidirectional pins that connect to the data bus on the microprocessor. |
| IRQ | (Interrupt Request)—an output that becomes a logic 1 whenever a key is pressed on the keyboard. This signal indicates that keyboard data are available for the microprocessor. |
| OUTA3–OUTA0 | (Most Significant Outputs)—used to send data to the displays. |
| OUTB3–OUTB0 | (Least Significant Outputs)—used to send data to the displays. |
| $\overline{RD}$ | (Read)—an input directly connected to the $\overline{IORC}$ or $\overline{RD}$ signal from the system. The $\overline{RD}$ input causes a read from the data registers or status register when $\overline{CS}$ is a logic 0. |
| RESET | (Reset)— an input that connects to the system RESET signal. |
| RL7–RL0 | (Return Lines)—inputs used to sense any key depression in the keyboard matrix. |
| SHIFT | (Shift)—an input normally connected to the shift key on a keyboard. |
| SL3–SL0 | (Scan Lines)—outputs that scan both the keyboard and the displays. |
| $\overline{WR}$ | (Write)—an input, developed with external logic, that connects to the write strobe signal. The $\overline{WR}$ inputs cause data to be written to either the data registers or control registers within the 8279. |
| Vcc | (Supply)—a pin connected to the system +5.0 V bus. |
| Vss | (Ground)—a pin connected to the system ground. |

## Interfacing the 8279 to the Microprocessor

In Figure 11–26, the 8279 is connected to the 8088 microprocessor. The 8279 is decoded to function at 8-bit I/O addresses 10H and 11H, where 10H is the data port and 11H is the control port. This circuit uses a PAL16L8 (see Example 11–16) to decode the I/O address and to generate the low bank I/O write signal for the 8279. Address bus bit A0 selects either the data or control port. Notice that the CS signal selects the 8279 and provides a signal called $\overline{WAIT2}$, used to cause 2 wait states, so this device can function with an 8MHz 8088.

**EXAMPLE 11–16**

```
TITLE          Address Decoder
PATTERN        Test 9
REVISION       A
AUTHOR         Barry B. Brey
COMPANY        Symbiotic Systems
DATE           4/30/94
CHIP           Decoder9 PAL16L8
```

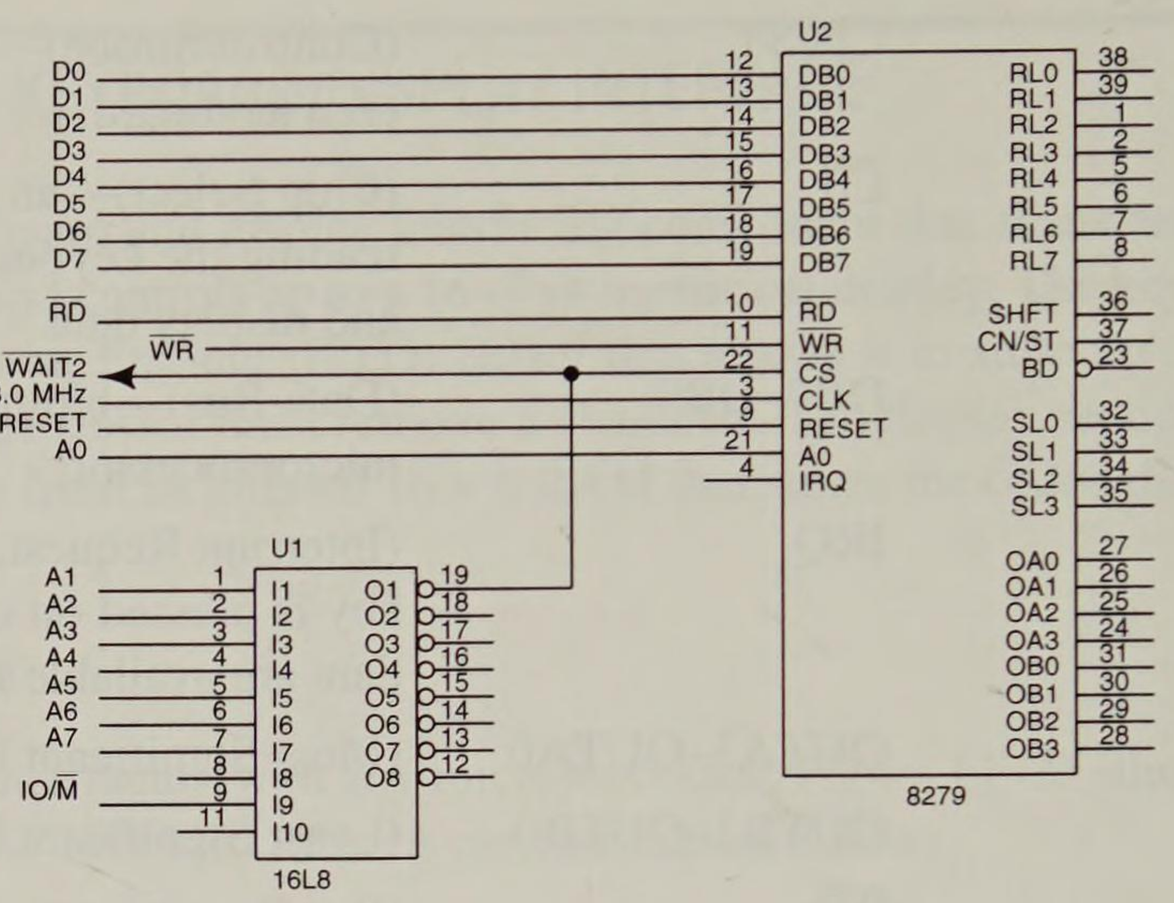



**FIGURE 11–26** The 8279 interfaced to the 8088 microprocessor to function at 8-bit I/O ports 10H and 11H.

```
;pins     1  2  3  4  5  6  7  8  9   10
          A1 A2 A3 A4 A5 A6 A7 NC IOM GND

;pins     11 12 13 14 15 16 17 18 19 20
          NC NC NC NC NC NC NC NC CS VCC

EQUATIONS

/CS = IOM * /A7 * /A6 * /A5 * A4 * /A3 * /A2 * /A1
```

The only signal not connected to the microprocessor is the IRQ output. This is an interrupt request pin and is beyond the scope of this section of the text. The next chapter explains the operation and function of interrupts in a system.

## Keyboard Interface

Suppose that a 64-key keyboard (with no numeric displays) is connected, through the 8279, to the 8088 microprocessor. Figure 11–27 shows this connection as well as the keyboard. With the 8279, the keyboard matrix is any size between $2 \times 2$ (4 keys) and $8 \times 8$ (64 keys). (Note that each crossover point in the matrix contains a normally open push-button switch that connects one vertical column with one horizontal row when a key is pressed.) The I/O port number decoded is the same as that decoded for Figure 11–26. The I/O port number is 10H for the data port and 11H for the control port in this circuit.

The 74ALS138 decoder generates eight active low column strobe signals for the keyboard. The selection pins SL2–SL0 sequentially scan each column of the keyboard, and the internal circuitry of the 8279 scans the RL pins searching for a key switch closure. Pullup resisters, normally found on input lines of a keyboard, are not required, because the 8279 contains its own internal pullups on the RL inputs.

*Programming the Keyboard Interface.* Before any keystroke is detected, the 8279 must be programmed—a more involved procedure than programming the 8255. The 8279 has eight control words to consider before it is programmed. The first 3 bits of the number sent to the control port (11H, in this example) each select one of the 8 different control words. Table 11–3 lists all eight control words and briefly describes them.

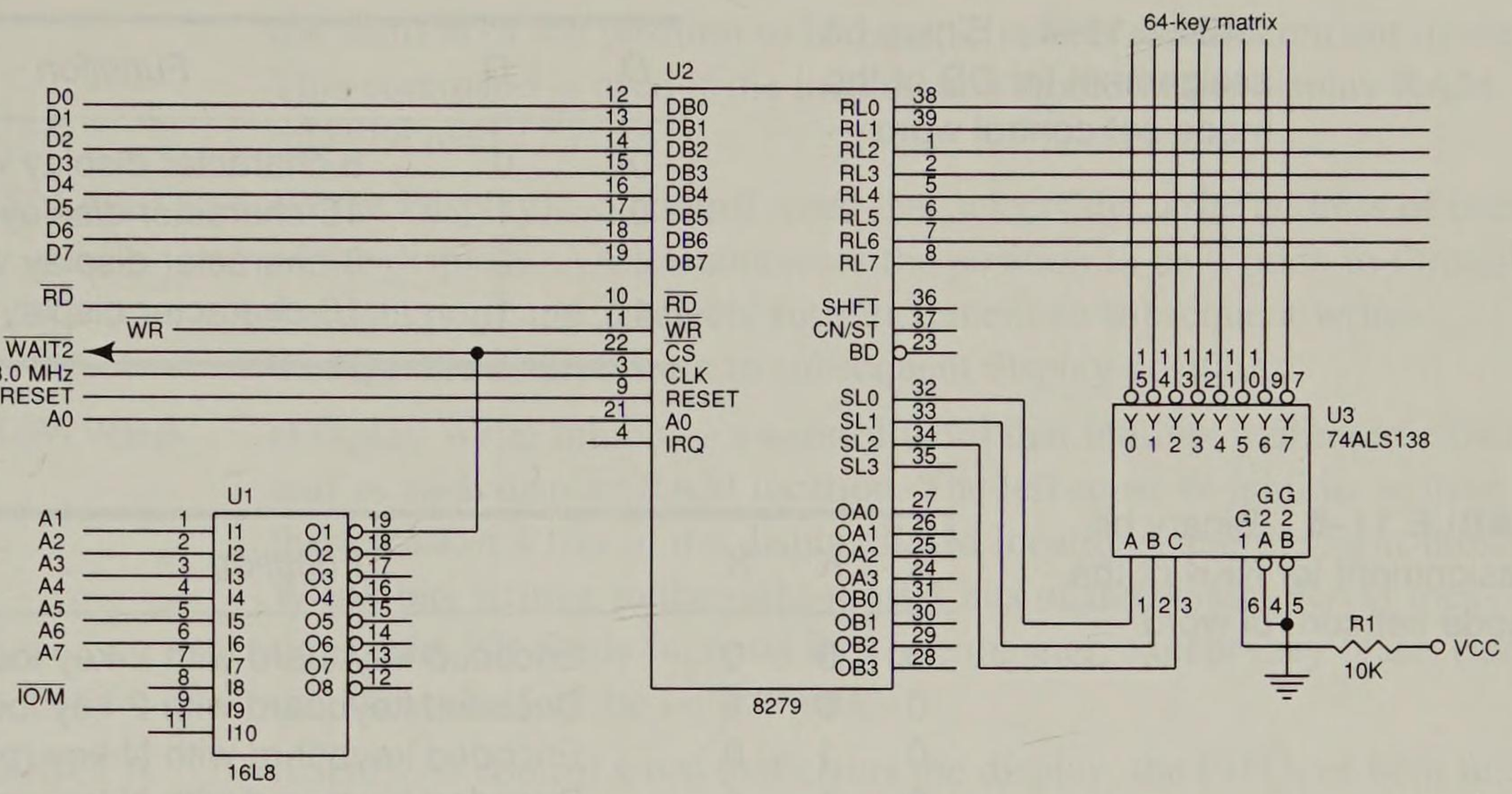



**FIGURE 11–27** A 64-key keyboard interfaced to the 8088 microprocessor through the 8279.

**TABLE 11–3** The 8279 control word summary

| *D7* | *D6* | *D5* | *Function* | *Purpose* |
|---|---|---|---|---|
| 0 | 0 | 0 | Mode set | Selects the number of display positions, left or right entry, and type of keyboard scan |
| 0 | 0 | 1 | Clock | Programs the internal clock and sets the scan and debounce times |
| 0 | 1 | 0 | Read FIFO | Selects the type of FIFO read and the address to be read |
| 0 | 1 | 1 | Read display | Selects the type of display read and the address of the display position |
| 1 | 0 | 0 | Write display | Selects the type of display write and the address of the display position |
| 1 | 0 | 1 | Display write inhibit | Allows half-bytes to be blanked or inhibited |
| 1 | 1 | 0 | Clear | Allows the display or FIFO to be cleared |
| 1 | 1 | 1 | End interrupt | Clears the IRQ pin at the end of an interrupt service procedure |

*Control Word Descriptions.* Following is a list of the control words that program the 8279. Note that the first three bits are the control register number, which are followed by other binary bits of information as they apply to each control.

000DDKKK (Mode Set)—a command with an opcode of 000 and two fields programmed to select the mode of operation for the 8279. The DD field selects the mode of operation for the displays (see Table 11–4) and the KKK field selects the mode of operation for the keyboard (see Table 11–5).

**TABLE 11–4** Binary bit assignment for DD of the mode set control word

| *D* | *D* | *Function* |
|---|---|---|
| 0 | 0 | 8-character display with left entry |
| 0 | 1 | 16-character display with left entry |
| 1 | 0 | 8-character display with right entry |
| 1 | 1 | 16-character display with right entry |

**TABLE 11–5** Binary bit assignment for KKK of the mode set control word

| *K* | *K* | *K* | *Function* |
|---|---|---|---|
| 0 | 0 | 0 | Encoded keyboard with 2-key lockout |
| 0 | 0 | 1 | Decoded keyboard with 2-key lockout |
| 0 | 1 | 0 | Encoded keyboard with N-key rollover |
| 0 | 1 | 1 | Decoded keyboard with N-key rollover |
| 1 | 0 | 0 | Encoded sensor matrix |
| 1 | 0 | 1 | Decoded sensor matrix |
| 1 | 1 | 0 | Strobed keyboard, encoded display scan |
| 1 | 1 | 1 | Strobed keyboard, decoded display scan |

The DD field selects either an 8- or 16-digit display and determines whether new data are entered to the right-most or left-most display position. The KKK field is quite a bit more complex. It provides encoded, decoded, or strobed keyboard operation.

In encoded mode, the SL outputs are active-high and follow the binary bit pattern 0 through 7 or 0 through 15, depending whether 8- or 16-digit displays are selected. In decoded mode, the SL outputs are active-low, and only one of the four outputs is low at any given instant. The decoded outputs repeat the pattern: 1110, 1101, 1011, and 0111. In strobed mode, an active-high pulse on the CN/ST input pin strobes data from the RL pins into an internal FIFO, where they are held for the microprocessor.

It is also possible to select either 2-key lockout or N-key rollover. Two-key lockout prevents two keys from being recognized if pressed simultaneously. N-key rollover will accept all keys pressed simultaneously.

001PPPPP (Clock)—a command word that programs the internal clock divider. The code PPPPP is a prescaler that divides the clock input pin (CLK) to achieve the desired operating frequency, approximately 100 KHz. An input clock of 1 MHz thus requires a prescaler of $01010_2$ for PPPPP.

010Z0AAA (Read FIFO)—a control word that selects the address of a keystroke from the internal FIFO buffer. Bit positions AAA select the desired FIFO location from 000 to 111, and Z selects auto-increment for the address. Under normal operation, this control word is used only with the sensor matrix operation of the 8279.

011ZAAAA (Display Read)—a control word that selects the read address of one of the display RAM positions for reading through the data port. AAAA is

the address of the position to be read; Z selects auto-increment mode. This command is used if the information stored in the display RAM must be read.

100ZAAAA (Write Display)—a control word that selects the write address of one of the displays. AAAA addresses the position to be written to through the data port, and Z selects auto-increment so subsequent writes through the data port are to subsequent display positions.

1010WWBB (Display Write Inhibit)—a control word that inhibits writing to either half of each display RAM location. The left-most W inhibits writing to the left-most 4 bits of the display RAM locations, and the right-most W inhibits writing to the right-most 4 bits of the display RAM locations. The BB fields function in a like manner, except they blank (turn off) either half of the output pins.

1100CCFA (Clear)—a control word that clears the display, the FIFO, or both the display and FIFO. Bit F clears the FIFO and the display RAM status and sets the address pointer to 000. If the CC bits are 00 or 01, all display RAM locations become 0000000; if CC = 10, all locations become 00100000; if CC = 11, all locations become 11111111.

111E000 (End Interrupt)—a control word issued to clear the IRQ pin to zero in the sensor matrix mode.

The large number of control words make programming the keyboard interface appear complex. Before anything is programmed, the clock divider rate must be determined. In the circuit illustrated in Figure 11–26, we use a 3.0 MHz clock input signal. To program the prescaler to generate a 100 KHz internal rate, we program PPPPP of the clock control word with a 30, or $11110_2$.

The next step involves programming the keyboard type. In Figure 11–27, we have an encoded keyboard. The circuit includes an external decoder that converts the encoded data from the SL pins into decoded column selection signals. We are free in this example to choose either 2-key lockout or N-key rollover, but most applications use 2-key lockout.

Finally, we program the operation of the FIFO. Once the FIFO is programmed, it never needs to be reprogrammed unless we need to read prior keyboard codes. Each time a key is typed, the data are stored in the FIFO; if data are read from the FIFO before the FIFO is full (8 characters), then the data from the FIFO follow the same order as the typed data. Example 11–17 provides the software required to initialize the 8279 to control the keyboard illustrated in Figure 11–27.

**EXAMPLE 11–17**

```
                        ;Initialization dialog for the keyboard interface
                        ;
                        ;program clock

0000 B0 3E                      MOV  AL,00111110B
0002 E6 11                      OUT  CNTR,AL

                        ;program mode set

0004 B0 00                      MOV  AL,0
```

```
0006 E6 11                      OUT   CNTR,AL

                          ;program to read FIFO and keyboard

0008 B0 50                      MOV   AL,01010101B
000A E6 11                      OUT   CNTR,AL
```

Once the 8279 is initialized, a procedure is required to read data from the keyboard. We determine whether a character is typed in the keyboard by looking at the FIFO status register. Whenever the control port is addressed by the IN instruction, the contents of the FIFO status word are copied into the AL register. Figure 11–28 shows the contents of the FIFO status register and defines the purpose of each status bit.

The procedure listed in Example 11–18 first tests the FIFO status register to see whether it contains any data. If NNN = 000, the FIFO is empty. Upon determining that the FIFO is not empty, the procedure inputs data to AL and returns with the keyboard code in AL.

**EXAMPLE 11–18**

```
                          ;Procedure that reads data from the FIFO and
                          ;returns with it in AL.
                          ;
= 0007                    MASKS  EQU  7

0000                      READ   PROC FAR

                          ;test FIFO status

0000 E4 11                      IN    AL,STATUS       ;get status
0002 A8 07                      TEST  AL,MASKS        ;test NNN
0004 74 FA                      JZ    READ            ;if NNN = 0

                          ;read FIFO

0006 E4 10                      IN    AL,DATA         ;get data
0008 CB                         RET

0009                      READ   ENDP
```

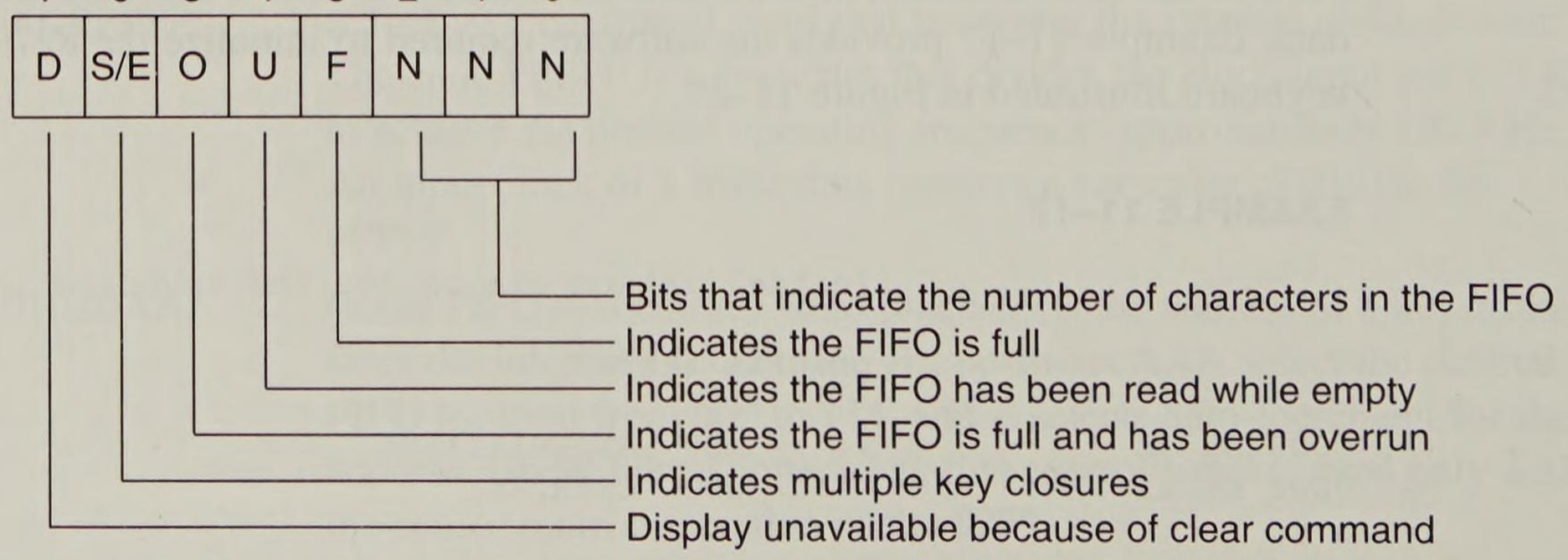



**FIGURE 11–28** The 8279-5 FIFO status register.

The data found in AL upon returning from the subroutine contain raw data from the keyboard. Figure 11–29 shows the format of this data for both the scanned and strobed modes of operation. The scanned code is returned from our keyboard interface and converted to ASCII code by using the XLAT instruction with an ASCII code lookup table. The scanned code is returned from our keyboard interface with the row and column number occupying the right-most 6 bits. The SH shows the state of the shift pin and the CT bit shows the state of the control pin. In the strobed mode, the contents of the eight RL inputs appear just as they are sampled, by placing a logic 1 on the strobe input pin to the 8279.

## Six-Digit Display Interface

Figure 11–30 depicts the 8279 connected to the 8088 microprocessor and a 6-digit numeric display. This interface uses a PAL16L8 (program not shown) to decode the 8279 at I/O ports 20H (data) and 21H (control/status). The segment data are supplied to the displays through the OUTA and OUTB pins of the 8279. These bits are buffered by a segment driver (ULN2003A), to drive the segment inputs to the display.

A 74ALS138 3-to-8 line decoder enables the anode switches of each display position. The SL2–SL0 pins supply the decoder with the encoded display position from the 8279. Notice that the left-hand display is at position 0101 and the right-hand display is at position 0000. These are the addresses of the display positions as indicated in control words for the 8279.

It is necessary to choose resister values that allow 60 mA of current flow per segment. In this circuit we use 47Ω resisters. If we allow 60 mA of segment current, then the average segment current is one-sixth of 60 mA, or 10 mA, because current only flows through a segment for one-sixth of the time. The anode switches must supply the current for all seven segments plus the decimal point. Here,the total anode current is $8 \times 60$ mA, or 480 mA.

Example 11–19 lists the initialization dialog for programming the 8279 to function with this 6-digit display. This software programs the display and clears the display RAM.

**EXAMPLE 11–19**

```
                        ;Initialization dialog for the 6 digit display
                        ;
                        ;program clock

0000 B0 3E                      MOV  AL,00111110B
0002 E6 21                      OUT  CNTR,AL

                              ;program mode set

0004 B0 00                      MOV  AL,0
0006 E6 21                      OUT  CNTR,AL
```

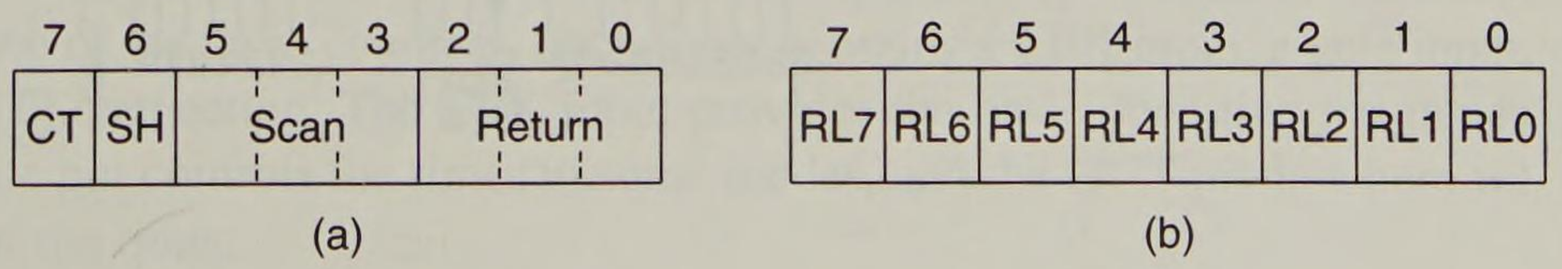



**FIGURE 11–29** The (a) scanned keyboard code and (b) strobed keyboard code for the 8279-5 FIFO.

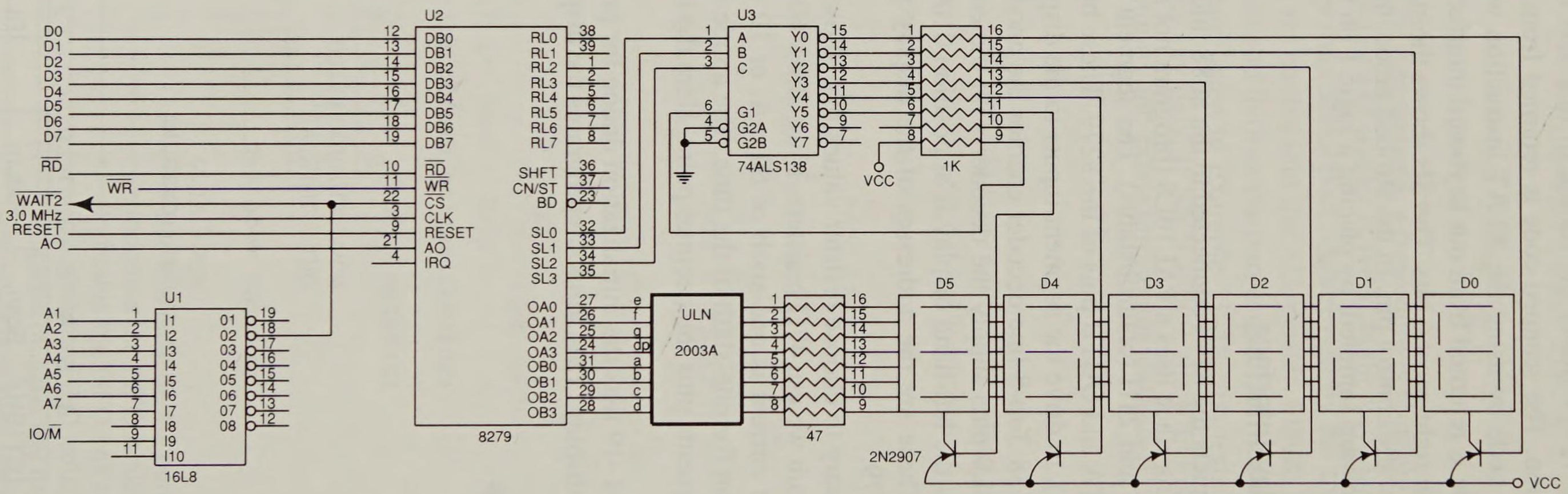



**FIGURE 11–30** A 6-digit numeric display interfaced to the 8279.

```
                        ;clear display

0008 B0 C1                      MOV   AL,11000001B
000A E6 21                      OUT   CNTR,AL
```

Example 11–20 lists a procedure for displaying information on the displays. Data are transferred to the procedure through the AX register. AH contains the seven-segment display code, and AL contains the address of the displayed digit.

**EXAMPLE 11–20**

```
                        ;Procedure to display AH on the display position
                        ;addressed by AL.
                        ;
= 0080                  MASKS   EQU   80H

000C                    DISP    PROC NEAR

000C 50                         PUSH AX                 ;save data
000D 0C 80                      OR   AL,MASKS           ;select display
000F E6 21                      OUT  CNTR,AL

                        ;display data

0011 8A C4                      MOV  AL,AH
0013 E6 20                      OUT  DATA,AL
0015 58                         POP  AX
0016 C3                         RET

0017                    DISP    ENDP
```

## 11–5 8254 PROGRAMMABLE INTERVAL TIMER

The 8254 programmable interval timer consists of three independent 16-bit programmable counters (timers). Each counter is capable of counting in binary or binary-coded decimal (BCD). The maximum allowable input frequency to any counter is 10 MHz. This device is useful when the microprocessor must control real-time events. Some examples of usage include: real-time clock, events counter, and motor speed and direction control.

This timer also appears in the personal computer, decoded at ports 40H–43H to (1) generate a basic timer interrupt that occurs at approximately 18.2 HZ, (2) cause the DRAM memory system to be refreshed, and (3) provide a timing source to the internal speaker and other devices. The timer in the personal computer is an 8253 instead of an 8254.

### 8254 Functional Description

Figure 11–31 shows the pinout of the 8254, which is a higher-speed version of the 8253, and a diagram of one of the three counters. Each timer contains a CLK input, a gate input, and an output (OUT) connection. The CLK input provides the basic operating frequency to the timer, the gate pin controls the timer in some modes, and the OUT pin is where we obtain the output of the timer.

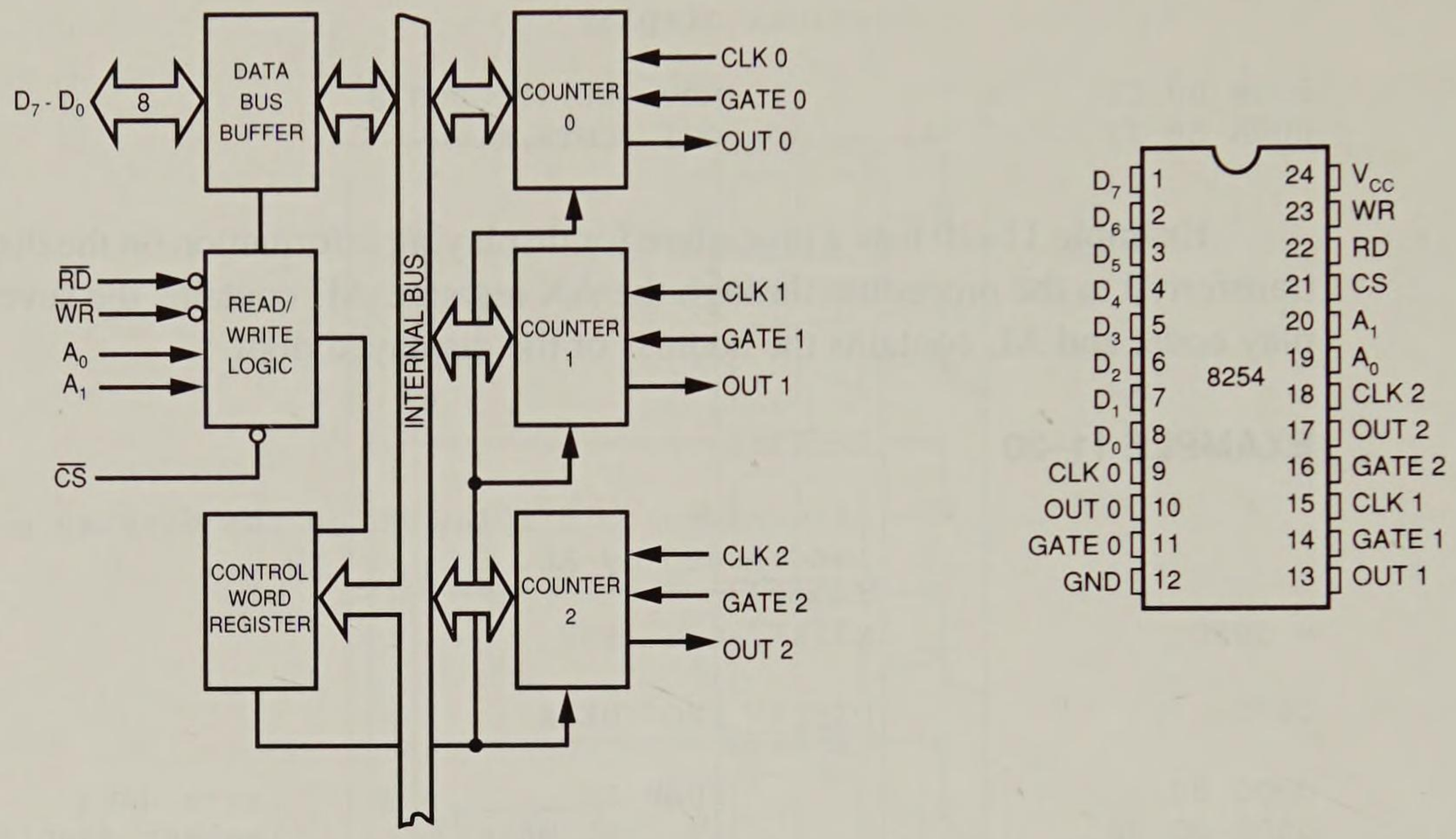



**FIGURE 11–31** The 8254 programmable interval timer. (a) Internal structure. (b) Pinout. (Courtesy of Intel Corporation)

The signals that connect to the microprocessor are the data bus pins (D7–D0), $\overline{RD}$, $\overline{WR}$, $\overline{CS}$, and address inputs A1 and A0. The address inputs are present to select any of the four internal registers used for programming, reading, or writing to a counter. The personal computer contains an 8253 timer, or its equivalent, decoded at I/O ports 40H–43H. Timer zero is programmed to generate an 18.2 Hz signal that interrupts the microprocessor at interrupt vector 8 for a clock tick. The tick is often used to time programs and events. Timer 1 is programmed for a 15 μs output, which is used on the PC/XT personal computer to request a DMA action to refresh the dynamic RAM. Timer 2 is programmed to generate tone on the personal computer speaker.

### *Pin Definitions*

A1, A0 (Address Inputs)—select one of four internal registers within the 8254. See Table 11–6 for the function of the A1 and A0 address bits.

CLK (Clock Input)—used as a timing source for each of the internal counters. This input is often connected to the PCLK signal from the microprocessor system bus controller.

**TABLE 11–6** Address selection inputs to the 8254

| *A1* | *A0* | *Function* |
|---|---|---|
| 0 | 0 | Counter 0 |
| 0 | 1 | Counter 1 |
| 1 | 0 | Counter 2 |
| 1 | 1 | Control word |

$\overline{CS}$ (Chip Select)—enables the 8254 for programming and reading or writing a counter.

G (Gate Input)—controls the operation of the counter in some modes of operation.

GND (Ground)—connected to the system ground bus.

OUT (Counter Output)—the output is where the waveform generated by the timer is available.

$\overline{RD}$ (Read)—reads data from the 8254 with the $\overline{IORC}$ signal.

Vcc (Power)—connected to the +5.0 V power supply.

$\overline{WR}$ (Write)—writes data to the 8254 with the write strobe ($\overline{IOWC}$).

## Programming the 8254

Each counter is individually programmed by writing a control word, followed by the initial count. Figure 11–32 lists the program control word structure of the 8254. The control word allows the programmer to select the counter, mode of operation, and type of operation (read/write). The control word also selects either a binary or BCD count. Each counter may be programmed with a count of 1 to FFFFH. A count of 0 is equal to FFFFH+1 (65,536), or 10,000 in BCD. The minimum count of 1 applies to all modes of operation except modes 2 and 3, which have a minimum count of 2. Timer 0 is used in the personal computer, with a divide by count of 64K (FFFFH) to generate the 18.2 Hz (18.19638 Hz) interrupt clock tick. Timer 0 has a clock input frequency of 4.77 MHz ÷ 4, or 1.1925 Mhz.

The control word uses the BCD bit to select a BCD count (BCD = 1) or a binary count (BCD = 0). The M2, M1, and M0 bits select one of the 6 different modes of operation (000–101) for the counter. The RW1 and RW0 bits determine how the data are read

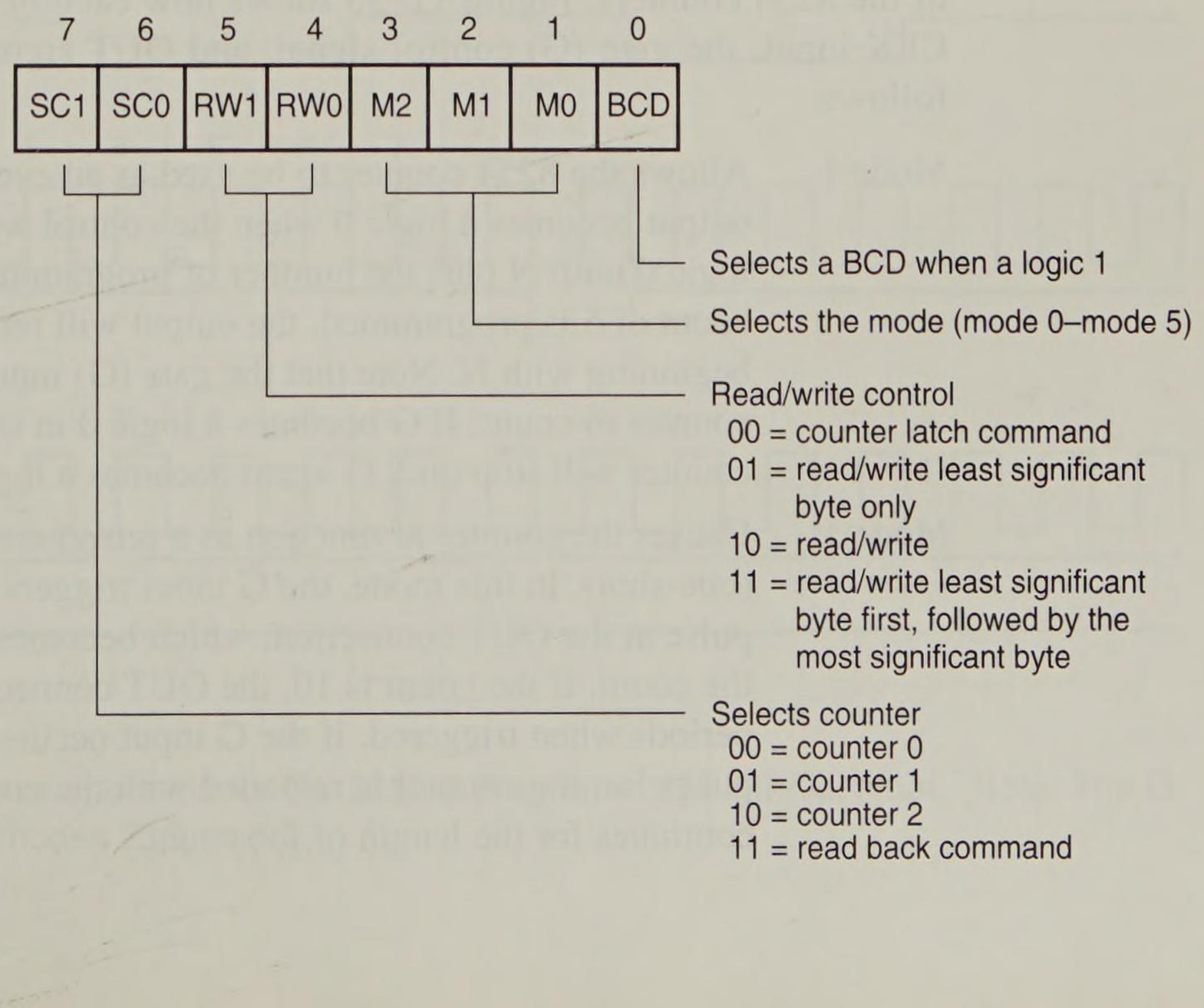



**FIGURE 11–32** The control word for the 8254-2 timer.

from, or written to, the counter. The SC1 and SC0 bits select a counter or the special read-back mode of operation discussed later in this section.

Each counter has a program control word used to select the way the counter operates. If 2 bytes are programmed into a counter, then the first byte (LSB) will stop the count, and the second byte (MSB) will start the counter with the new count. The order of programming is important for each counter, but programming of different counters may be interleaved for better control. For example, the control word may be sent to each counter before the counts for individual programming. Example 11–21 shows a few ways to program counters 1 and 2. The first method programs both control words, then the LSB of the count for each counters, which stops it from counting. Finally, the MSB portion of the count is programmed, starting both counters with the new count. The second example shows one counter programmed before the other.

**EXAMPLE 11–21**

```
PROGRAM CONTROL WORD 1          ;set up counter 1
PROGRAM CONTROL WORD 2          ;set up counter 2
PROGRAM LSB 1                   ;stop counter 1 and program LSB
PROGRAM LSB 2                   ;stop counter 2 and program LSB
PROGRAM MSB 1                   ;program MSB and start counter 1
PROGRAM MSB 2                   ;program MSB and start counter 2

   or

PROGRAM CONTROL WORD 1          ;set up counter 1
PROGRAM LSB 1                   ;stop counter 1 and program LSB
PROGRAM MSB 1                   ;program MSB and start counter 1
PROGRAM CONTROL WORD 2          ;set up counter 2
PROGRAM LSB 2                   ;stop counter 2 and program LSB
PROGRAM MSB 2                   ;program MSB and start counter 2
```

*Modes of Operation.* Six modes (mode 0–mode 5) of operation are available to each of the 8254 counters. Figure 11–33 shows how each of these modes functions with the CLK input, the gate (G) control signal, and OUT signal. A description of each mode follows:

Mode 1 — Allows the 8254 counter to be used as an events counter. In this mode, the output becomes a logic 0 when the control word is written, and remains logic 0 until N plus the number of programmed counts. For example, if a count of 5 is programmed, the output will remain a logic 0 for 6 counts, beginning with N. Note that the gate (G) input must be a logic 1 to allow the counter to count. If G becomes a logic 0 in the middle of the count, the counter will stop until G again becomes a logic 1.

Mode 1 — Causes the counter to function as a retriggerable monostable multivibrator (one-shot). In this mode, the G input triggers the counter so that it develops a pulse at the OUT connection, which becomes a logic 0 for the duration of the count. If the count is 10, the OUT connection goes low for 10 clocking periods when triggered. If the G input occurs within the duration of the output pulse, the counter is reloaded with the count and the OUT connection continues for the length of the count.

Mode 2 Allows the counter to generate a series of continuous pulses that are one clock pulse in width. The separation between pulses is determined by the count. For example, for a count of 10, the output is a logic 1 for 9 clock periods and low for 1 clock period. This cycle is repeated until the counter is programmed with a new count, or until the G pin is placed at a logic 0 level. The G input must be a logic 1 for this mode to generate a continuous series of pulses.

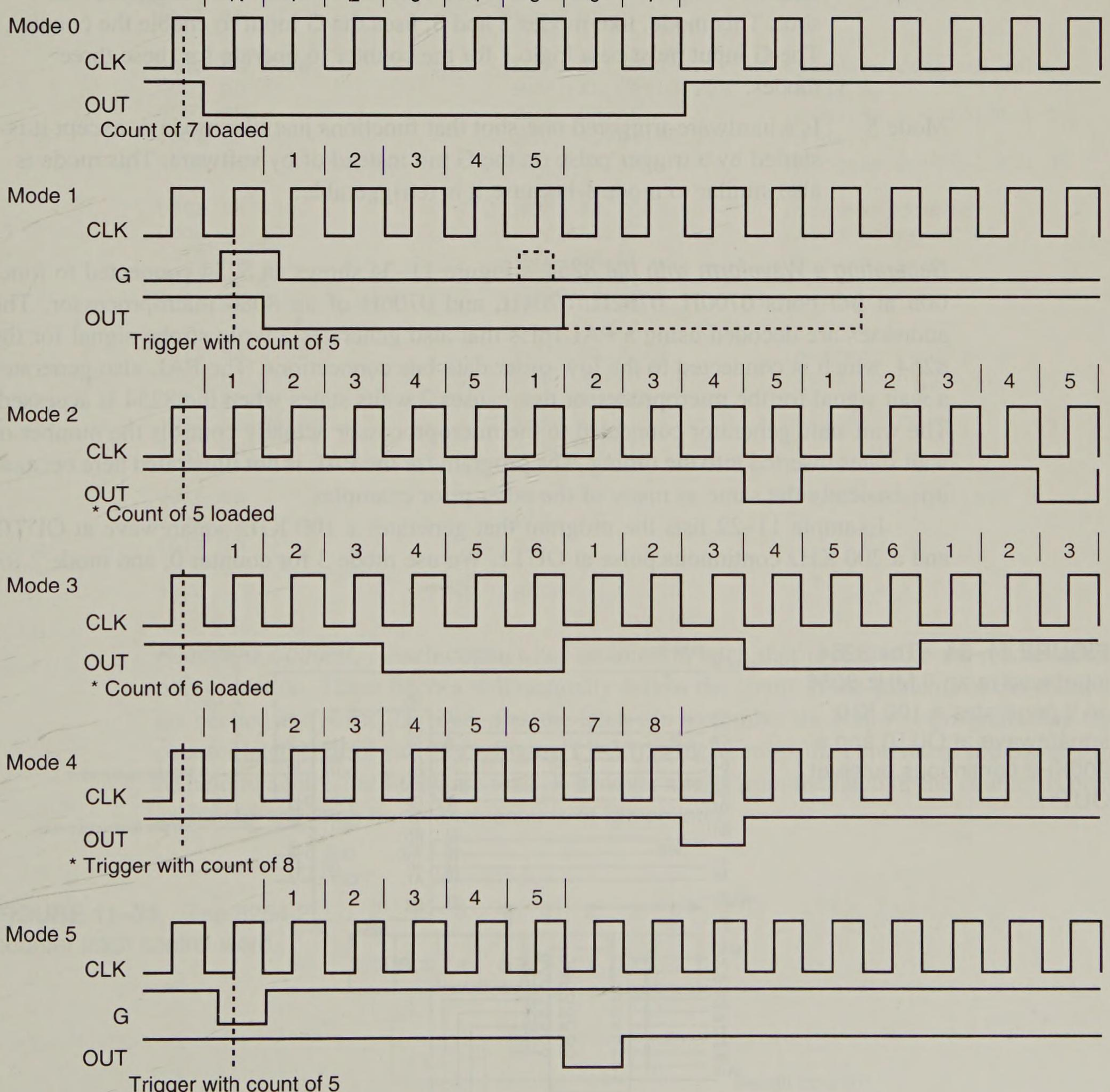



**FIGURE 11–33** The six modes of operation for the 8254-2 programmable interval timer. *Note: The G input stops the count when 0 in modes 2, 3, and 4.

Mode 3 Generates a continuous squarewave at the OUT connection, provided the G pin is a logic 1. If the count is even, the output is high for half the count and low for half the count. If the count is odd, the output is high for one clocking period longer than it is low. For example, if the counter is programmed for a count of 5, the output is high for 3 clocks and low for 2.

Mode 4 Allows the counter to produce a single pulse at the output. If the count is programmed as a 10, the output is high for 10 clocking periods and then low for one clocking period. The cycle does not begin until the counter is loaded with its complete count. This mode operates as a software-triggered one-shot. This mode, like modes 2 and 3, uses the G input to enable the counter. The G input must be a logic 1 for the counter to operate for these three modes.

Mode 5 Is a hardware-triggered one-shot that functions just like mode 4, except it is started by a trigger pulse on the G pin instead of by software. This mode is also similar to mode 1 because it is retriggerable.

*Generating a Waveform with the 8254.* Figure 11–34 shows an 8254 connected to function at I/O ports 0700H, 0702H, 0704H, and 0706H of an 8086 microprocessor. The addresses are decoded using a PAL16L8 that also generates a write strobe signal for the 8254, which is connected to the low-order data bus connections. The PAL also generates a wait signal for the microprocessor that causes 2 waits states when the 8254 is accessed. The wait state generator connected to the microprocessor actually controls the number of wait states inserted into the timing. The program for the PAL is not illustrated here because it is basically the same as many of the other prior examples.

Example 11–22 lists the program that generates a 100 KHz squarewave at OUT0, and a 200 KHz continuous pulse at OUT1. We use mode 3 for counter 0, and mode 2 for

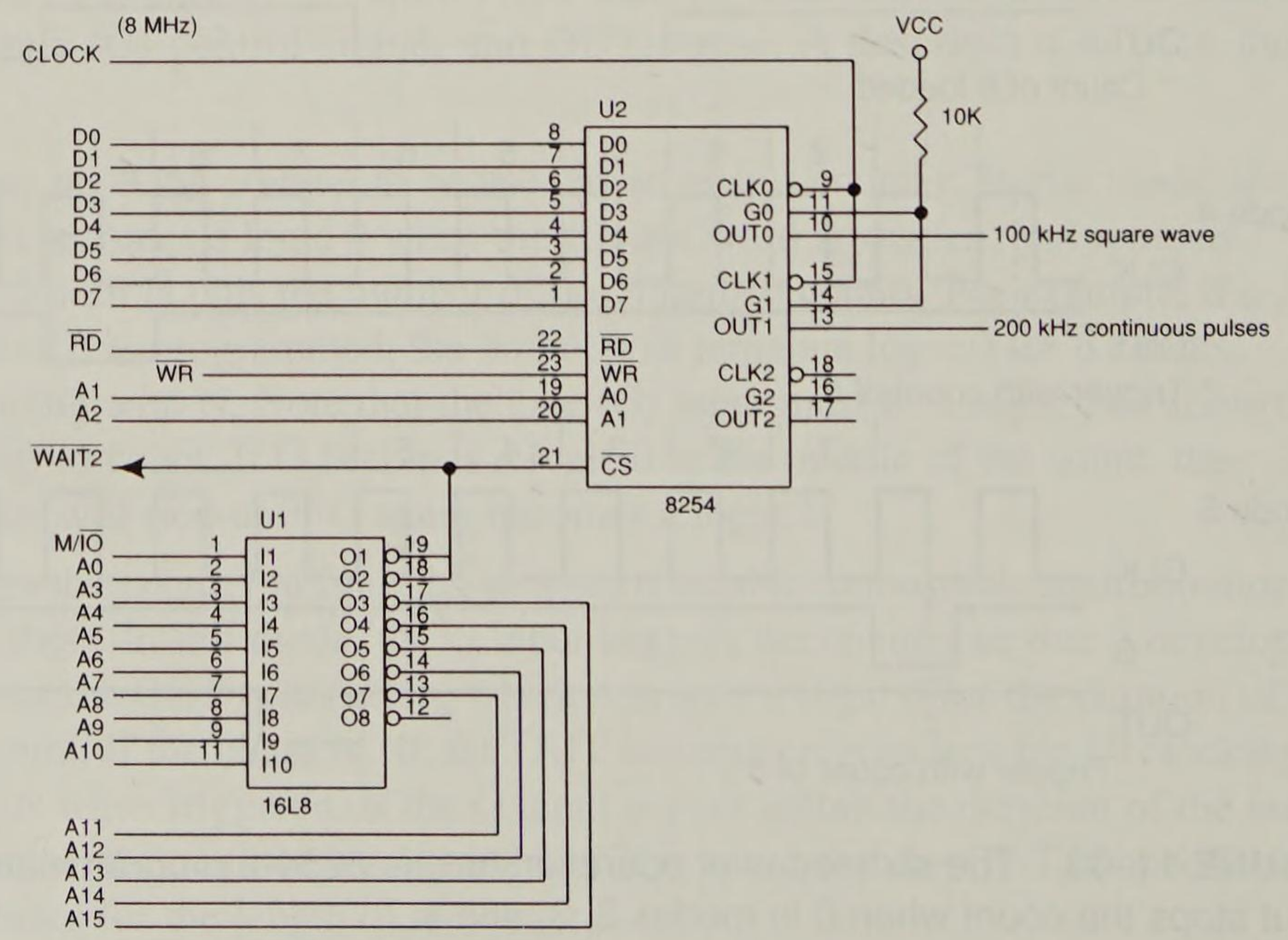



**FIGURE 11–34** The 8254 interfaced to an 8 MHz 8086 so it generates a 100 KHz squarewave at OUT0 and a 200KHz continuous pulse at OUT1.

counter 1. The count for counter 0 is 80, and the count for counter 1 is 40. These counts generate the desired output frequencies with an 8 MHz input clock.

**EXAMPLE 11–22**

```
                        ;Procedure that programs the 8254 timer to
                        ;function as indicated in Figure 11-34
                        ;
0000                    TIME    PROC NEAR

0000 50                         PUSH AX                 ;save registers
0001 52                         PUSH DX

0002 BA 0706                    MOV  DX,706H            ;address control word
0005 B0 36                      MOV  AL,00110110B       ;mode 3
0007 EE                         OUT  DX,AL              ;program control for 0
0008 B0 74                      MOV  AL,01110100B       ;mode 2
000A EE                         OUT  DX,AL              ;program control for 1

000B BA 0700                    MOV  DX,700H            ;address counter 0
000E B0 50                      MOV  AL,80              ;load count of 80
0010 EE                         OUT  DX,AL
0011 32 C0                      XOR  AL,AL
0013 EE                         OUT  DX,AL

0014 BA 0702                    MOV  DX,702H            ;address counter 1
0017 B0 28                      MOV  AL,40              ;load count of 40
0019 EE                         OUT  DX,AL
001A 32 C0                      XOR  AL,AL
001C EE                         OUT  DX,AL

001D 5A                         POP  DX                 ;restore registers
001E 58                         POP  AX
001F C3                         RET

0020                    TIME    ENDP
```

*Reading a Counter.* Each counter has an internal latch that is read with the read counter port operation. These latches will normally follow the count. If the contents of the counter are needed at a particular time, then the latch can remember the count by programming the counter latch control word (see Figure 11–35), which causes the contents of the counter to be held in a latch until they are read. Whenever a read from the latch or the counter is programmed, the latch tracks the contents of the counter.

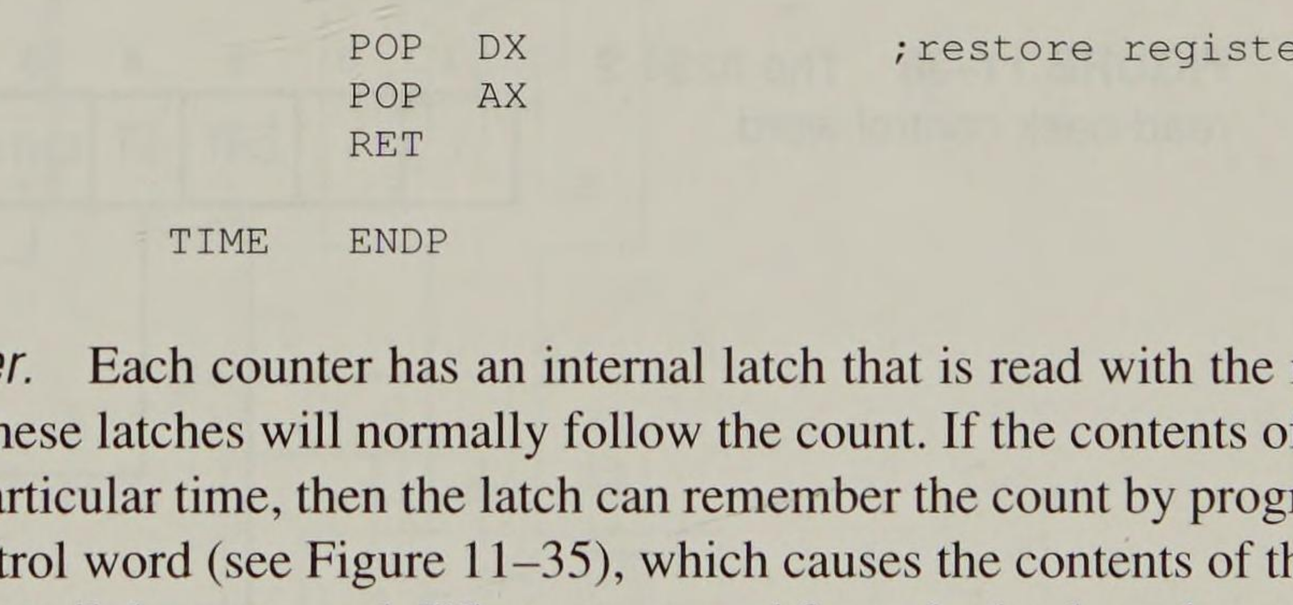



**FIGURE 11–35** The 8254-2 counter latch control word.

When it is necessary for the contents of more than one counter to be read at the same time, we use the read-back control word, illustrated in Figure 11–36. With the read-back control word, the $\overline{CNT}$ bit is a logic 0, which causes the counters selected by CNT0, CNT1, and CNT2 to be latched. If the status register is to be latched, then the $\overline{ST}$ bit is placed at a logic 0. Figure 11–37 shows the status register, which shows the state of the output pin, whether or not the counter is at its null state (0), and how the counter is programmed.

## DC Motor Speed and Direction Control

One application of the 8254 timer is as a motor speed controller for a DC motor. Figure 11–38 shows the schematic diagram of the motor and its associated driver circuitry. It also illustrates the interconnection of the 8254, a flip-flop, and the motor and its driver.

The operation of the motor driver circuitry is fairly straightforward. If the Q output of the 74ALS112 is a logic 1, the base Q2 is pulled up to +12 V through the base pullup resistor, and the base of Q2 is open circuited. This means that Q1 is off and Q2 is on, with ground applied to the positive lead of the motor. The bases of both Q3 and Q4 are pulled low to ground through the inverters. This causes Q3 to conduction or turn on, and Q4 to turn off, applying ground to the negative lead of the motor. The logic 1 at the Q output of the flip-flop, therefore, connects +12 V to the positive lead of the motor and applies ground to the negative lead. This connection causes the motor to spin in its forward direction. If the state of the Q output of the flip-flop becomes a logic 0, then the conditions of the transistors are reversed and +12 V is attached to the negative lead of the motor, with ground attached to the positive lead. This causes the motor to spin in the reverse direction.

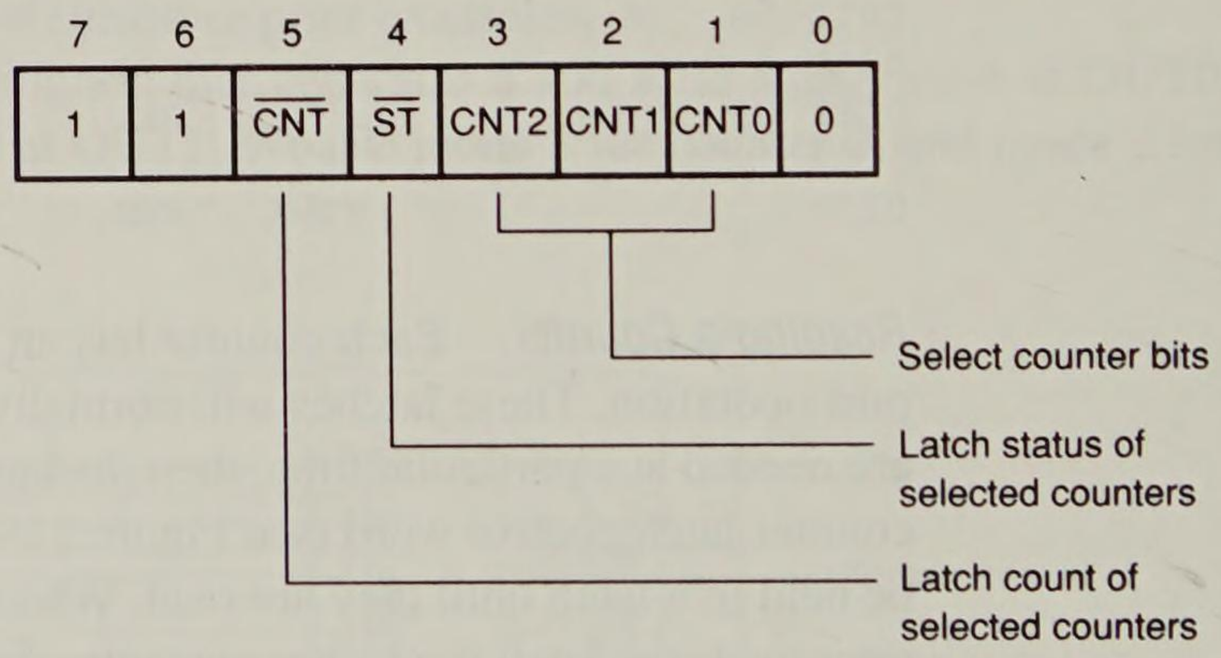



**FIGURE 11–36** The 8254-2 read-back control word.

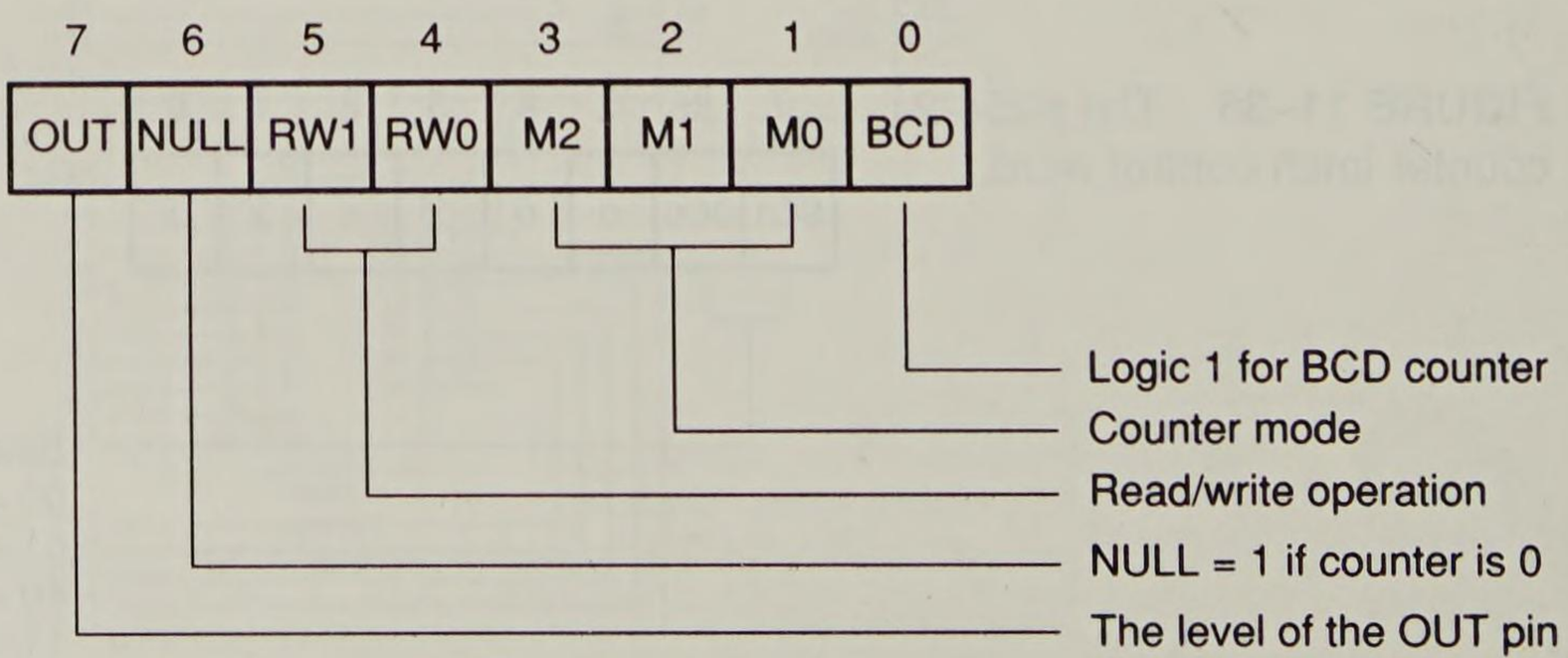



**FIGURE 11–37** The 8254-2 status register.

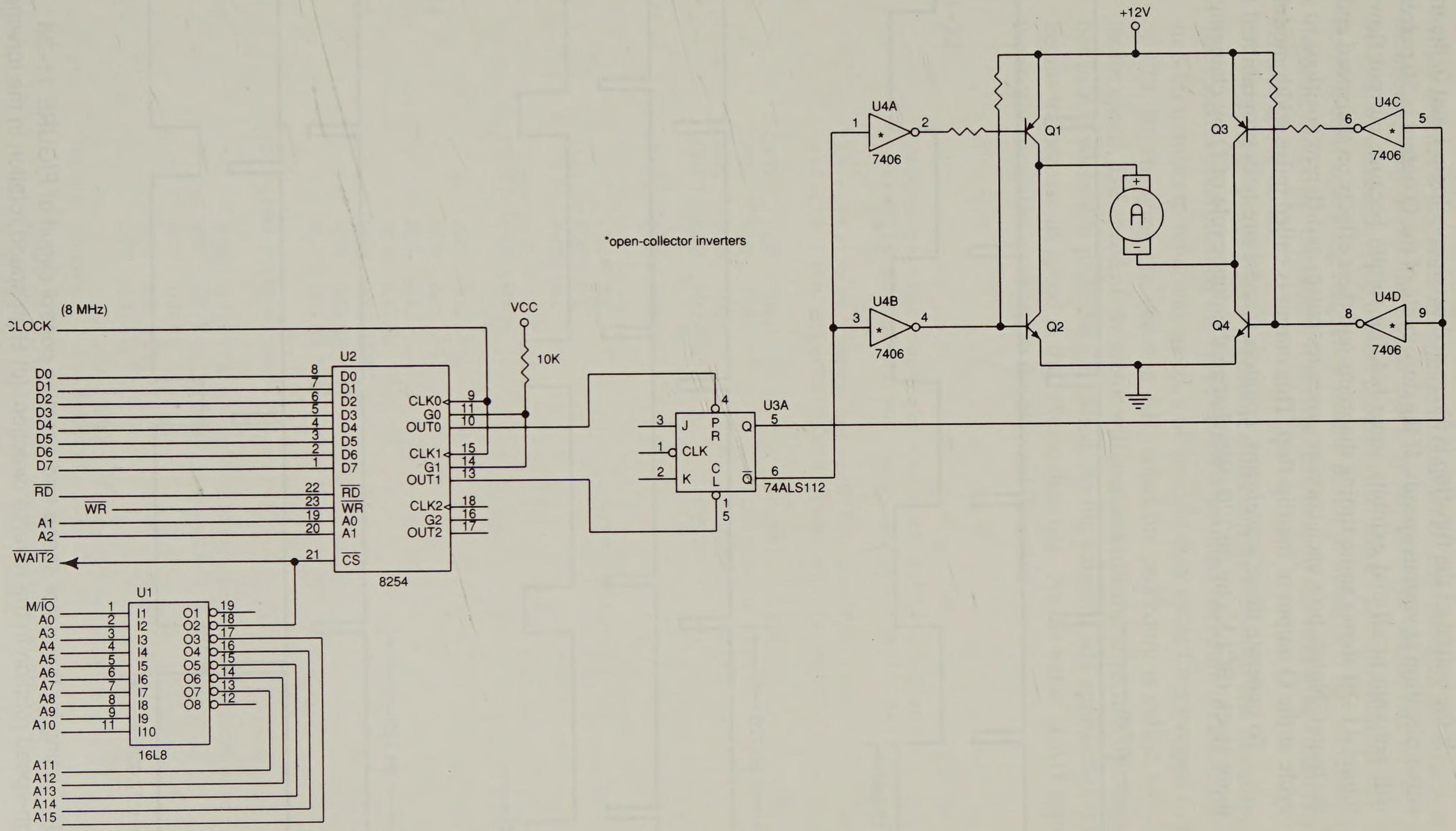



**FIGURE 11–38** Motor speed and direction control using the 8254 timer.

If the output of the flip-flop is alternated between a logic 1 and 0, the motor spins in either direction at various speeds. If the duty cycle of the Q output is 50 percent, the motor will not spin at all and exhibits some holding torque because current flows through it. Figure 11–39 shows some timing diagrams and their effects on the speed and direction of the motor. Notice how each counter generates pulses at different positions to vary the duty cycle at the Q output of the flip-flop. This output is called pulse width modulation.

To generate these waveforms, counters 0 and 1 are both programmed to divide the input clock (PCLK) by 30,720. We change the duty cycle of Q by changing the point at

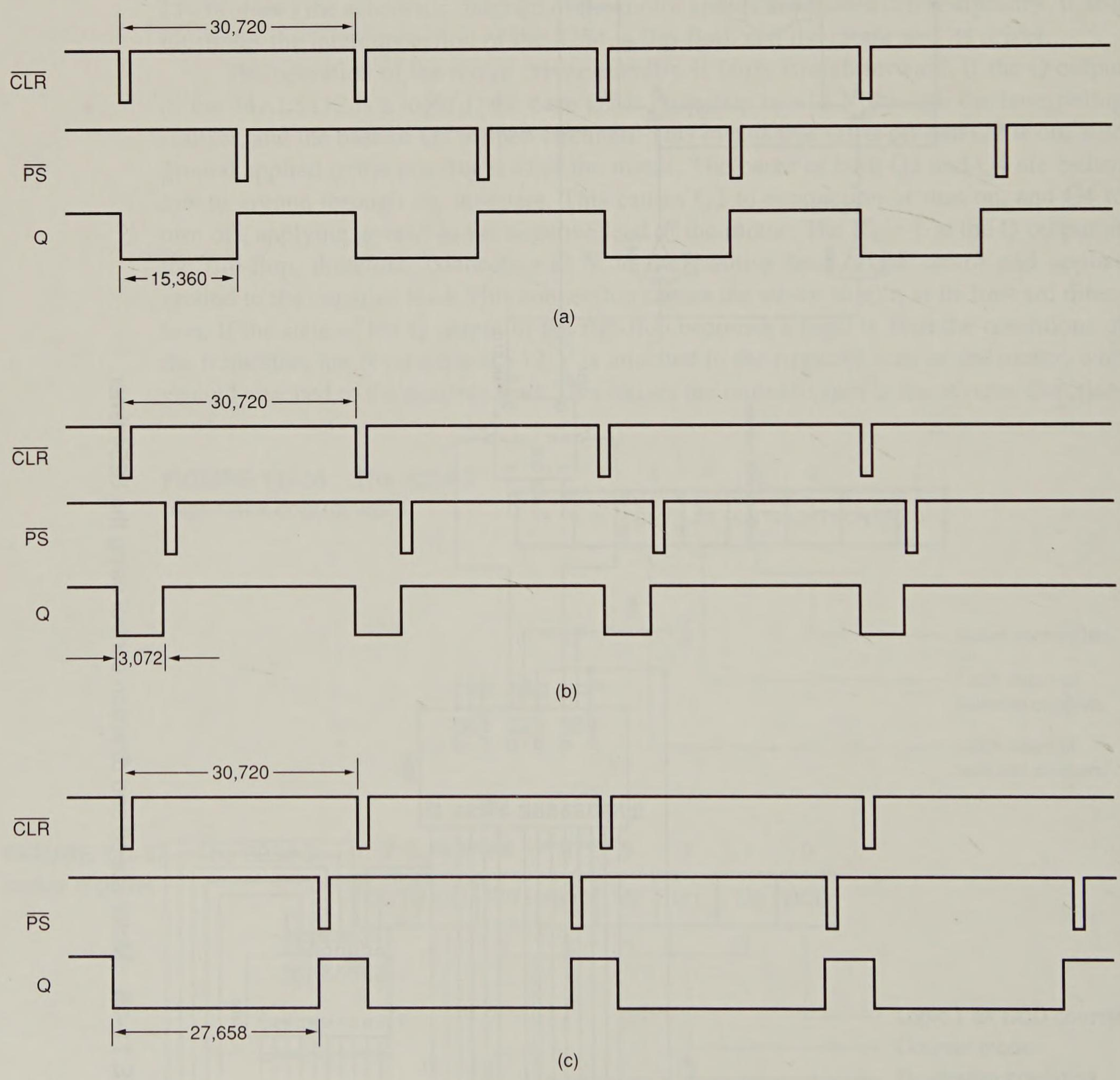



**FIGURE 11–39** Timing for the motor speed and direction control circuit of FIGURE 11–38 (a) No rotation. (b) High-speed rotation in the reverse direction. (c) High-speed rotation in the forward direction.

which counter 1 is started in relation to counter 0. This changes the direction and speed of the motor. Why divide the 8MHz clock by 30,720? The divide rate of 30,720 is divisible by 256, so we can develop a short program that allows 256 different speeds. This also produces a basic operating frequency for the motor of about 260 Hz, which is low enough in frequency to power the motor. It is important to keep this operating frequency below 1,000 Hz, but above 60 Hz.

Example 11–23 lists a procedure that controls the speed and direction of the motor. The speed is controlled by the value of AH when this procedure is called. Because we have an 8-bit number to represent speed, a 50 percent duty cycle for a stopped motor is a count of 128. By changing the value in AH, when the procedure is called, we can adjust the motor speed. The speed of the motor will increase in either direction by changing the number in AH when this procedure is called. As the value in AH approaches 00H, the motor begins to increase its speed in the reverse direction. As the value of AH approaches FFH, the motor increases its speed in the forward direction.

**EXAMPLE 11–23**

```
                        ;Procedure that controls the duty cycle of Q and
                        ;therefore the speed and direction of the motor.
                        ;
                        ;AH contains a number between 00H and FFH that
                        ;selects both direction and speed of the motor.
                        ;
= 0706                  CNTR    EQU  706H              ;control port
= 0700                  CNT0    EQU  700H              ;counter 0 port
= 0702                  CNT1    EQU  702H              ;counter 1 port
= 7800                  COUNT   EQU  30720             ;count of 30,720

0000                    SPEED   PROC NEAR

0000 50                         PUSH AX                ;save registers
0001 51                         PUSH DX
0002 53                         PUSH BX

                        ;calculate count

0003 8A DC                      MOV  BL,AH
0005 B8 0078                    MOV  AX,120
0008 F6 E3                      MUL  BL
000A 8B D8                      MOV  BX,AX
000C B8 7800                    MOV  AX,COUNT
000F 2B C3                      SUB  AX,BX
0011 8B D8                      MOV  BX,AX

                        ;program counter control words

0013 BA 0706                    MOV  DX,CNTR           ;load port address of control
0016 B0 34                      MOV  AL,00110100B      ;control for CNT0
0018 EE                         OUT  DX,AL
0019 B0 74                      MOV  AL,01110100B      ;control for CNT1
001B EE                         OUT  DX,AL

                        ;start counter 1 to generate clear

001C BA 0702                    MOV  DX,CNT1           ;address counter 1
001F B8 7800                    MOV  AX,COUNT          ;get count
```

```
0022 EE                         OUT  DX,AL          ;stop counter 1
0023 8A C4                      MOV  AL,AH
0025 EE                         OUT  DX,AL          ;start counter 1

                              ;wait for counter 1 to reach calculated count

0026                          SPE:

0026 EC                         IN   AL,DX          ;get count
0027 86 C4                      XCHG AL,AH
0029 EC                         IN   AL,DX
002A 86 C4                      XCHG AL,AH
002C 3B C3                      CMP  AX,BX          ;test count
002E 72 F6                      JB   SPE            ;if CNT1 below count

                              ;start counter 0 to generate set

0030 BA 0700                    MOV  DX,CNT0        ;address counter 0
0033 B8 7800                    MOV  AX,COUNT       ;get count
0036 EE                         OUT  DX,AL          ;stop counter 0
0037 8A C4                      MOV  AL,AH
0039 EE                         OUT  DX,AL          ;start counter 1

003A 5B                         POP  BX             ;restore registers
003B 5A                         POP  DX
003C 58                         POP  AX
003D C3                         RET

003E                          SPEED  ENDP
```

The procedure adjusts the waveform at Q by first calculating the count that counter 0 is to start in relationship to counter 1. This is accomplished by multiplying AH by 120 and then subtracting the product from 30,720. This is required because the counters are down-counters, which count from the programmed count to 0 before restarting. Next, counter 1 is programmed with a count of 30,720 and started, to generate the clear waveform for the flip-flop. After counter 1 is started it is read and compared with the calculated count. Once counter 1 reaches the calculated count, counter 0 is started with a count of 30,720. From this point forward, both counters continue generating the clear and set waveforms until the procedure to adjust the speed and direction of the motor is called again.

## 11–6 8251A PROGRAMMABLE COMMUNICATIONS INTERFACE

The 8251A is a programmable communications interface designed to connect to virtually any type of serial interface. The 8251A is a *universal synchronous/asynchronous receiver/transmitter* (USART) that is fully compatible with the Intel microprocessors, provided that 2 wait states are inserted if operated at 8 MHz. For connection to higher speed microprocessors, additional wait states must be inserted. The 8251A is capable of operating at 0–64 KBaud (Bd) (bits per second) in the synchronous mode, and 0–19.2 KBd in the asynchronous mode. *Baud rate* is the number of bits transferred per second, including start, stop, data, and parity. The programmer of the 8251A selects the number of data bits, number of stop bits, type of parity (even or odd), and the clock rate in the asynchronous mode.

In the synchronous mode, the programmer selects the number of data bits, parity, and the number of synchronization characters (one or two).

## Asynchronous Serial Data

Asynchronous serial data are information that is transmitted and received without a clock or timing signal. Figure 11–40 illustrates 2 frames of asynchronous serial data. Each frame contains a start bit, seven data bits, parity, and one stop bit. In this figure, a frame, which contains one ASCII character, has 10 bits. Most communications systems use 10 bits for asynchronous serial data with even parity.

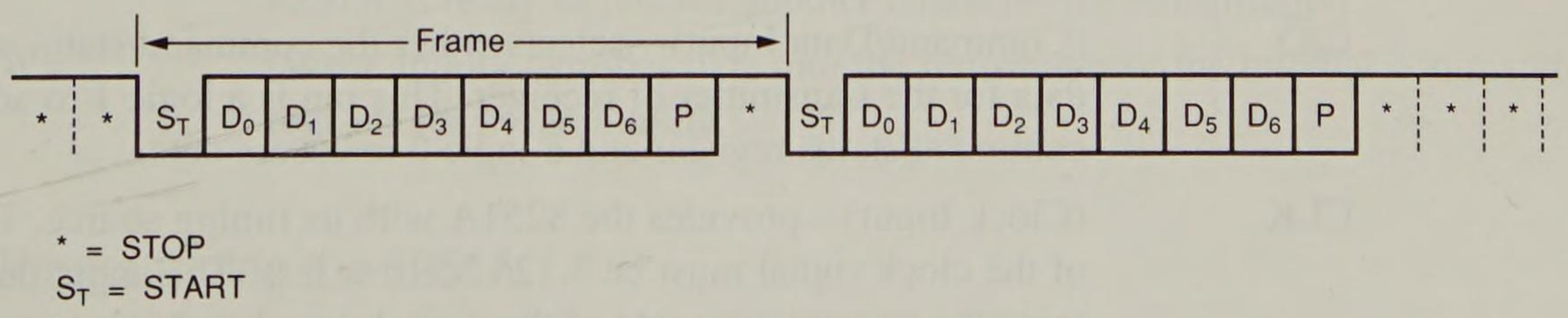
## Synchronous Serial Data

Figure 11–41 shows the format of synchronous serial data. Notice that this information contains no start or stop bits, only data and parity. Also notice that the data are referenced or synchronized to a clock or timing element. Instead of using the start and stop bits for synchronizing each character, synchronous data use a sync character or characters to synchronize blocks of data. A sync character or characters are sent, followed by a block of data in a synchronous system. The most common synchronous system uses Z sync characters, and is called *bi-sync*. The advantage of the synchronous system is that more bits of information are transferred per second, but the disadvantage is that the clock signal must also be transmitted with the data.

## 8251A Functional Description

Figure 11–42 illustrates the pinout of the 8251A USART. Two completely separate sections are responsible for data communications: the receiver and the transmitter. Because these sections are independent of each other, the 8251A is able to function in *simplex*, *half-duplex*, or *full-duplex* modes.

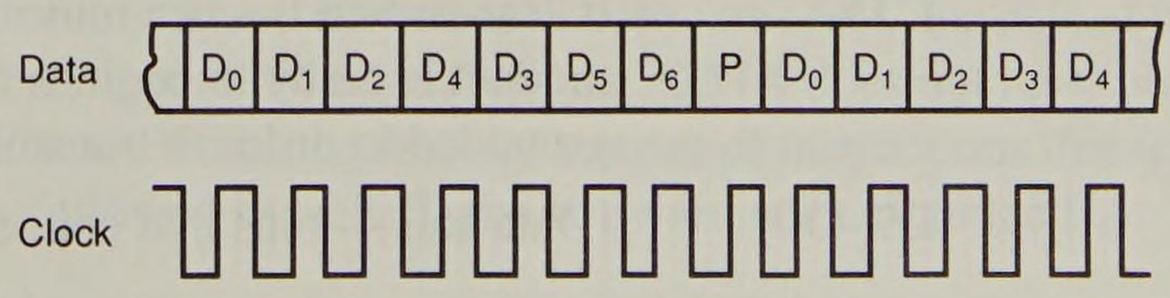
An example of a simplex system is where the transmitter or receiver is used by itself, such as in an FM (frequency modulation) radio station. An example half-duplex system is




* = STOP
$S_T$ = START

**FIGURE 11–40** Asynchronous serial data.

**FIGURE 11–41**
Synchronous serial data.



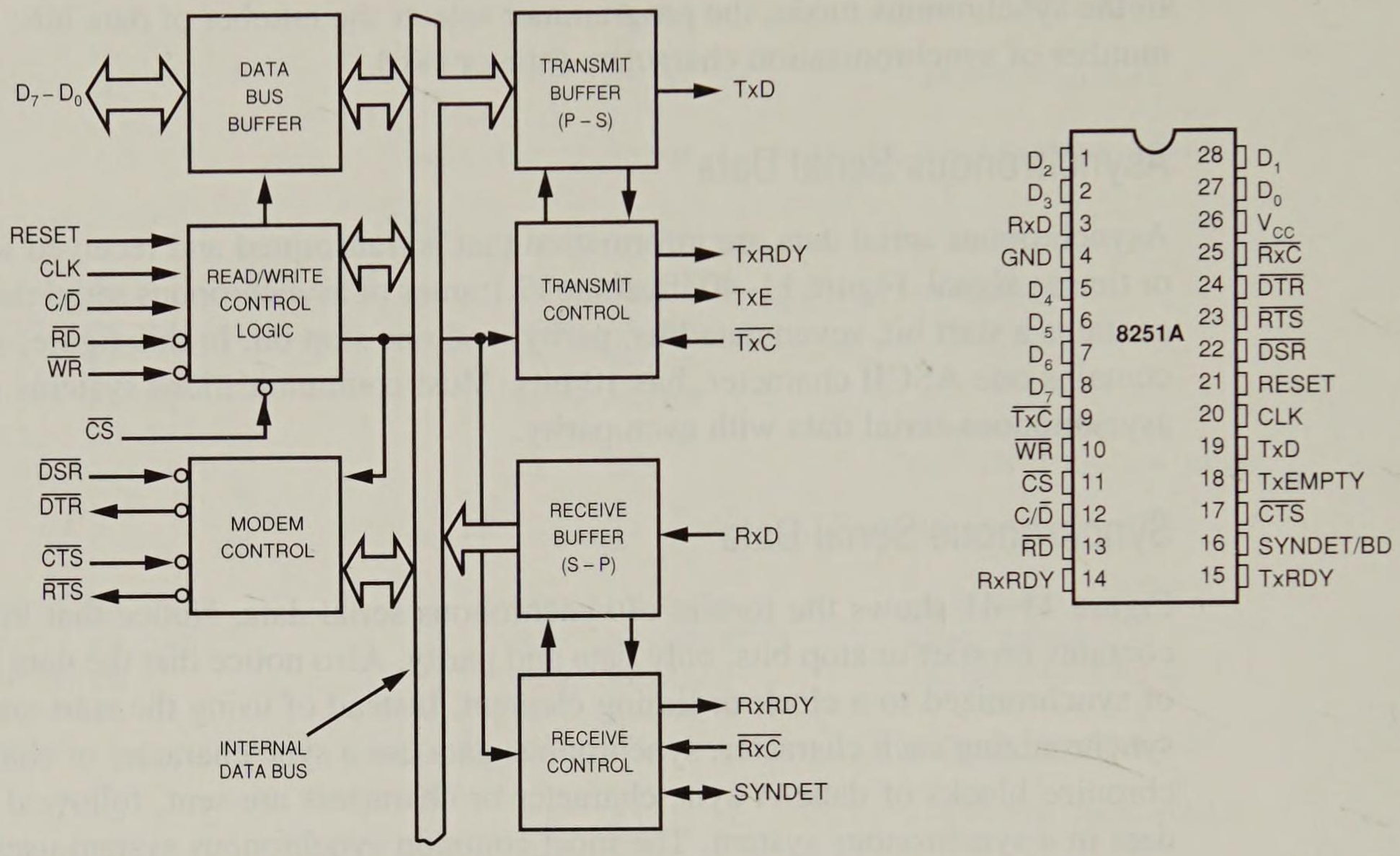



**FIGURE 11–42** The pinout and block diagram of the 8251A programmable communications interface. (Courtesy of Intel Corporation)

a CB (citizens band) radio, where we transmit and receive, but not both at the same time. The full-duplex system allows transmission and reception in both directions simultaneously. An example full-duplex system is the telephone.

The 8251A can control a *modem* (modulator/demodulator), which is a device that converts TTL levels of serial data into audio tones that can pass through the telephone system. Four pins on the 8251A are devoted to modem control: $\overline{\text{DSR}}$ (data set ready), $\overline{\text{DTR}}$ (data terminal ready), $\overline{\text{CTS}}$ (clear-to-send), and $\overline{\text{RTS}}$ (request-to-send). The modem is referred to as the *data set*, and the 8251A is referred to as the *data terminal*.

### *Pin Functions*

C/$\overline{\text{D}}$ (Command/Data Input)—selects either the command/status register or data for the transmitter or receiver. This pin is a logic 1 to access the command/status register and a logic 0 to access data.

CLK (Clock Input)—provides the 8251A with its timing source. The frequency of the clock signal must be 3.125 MHz or less. This input does not determine the transmission rate of the digital data, but that rate must be at least 16 times higher than the transmit or receive clocks.

$\overline{\text{CS}}$ (Chip Select)—enables the 8251A when a logic 0.

$\overline{\text{CTS}}$ (Clear-to-Send)—is a return signal from the modem that indicates the modem is ready to begin sending information. Note that this pin must be grounded in order to transmit data.

D7–D0 (Data Bus)—pins that connect to the data bus.

$\overline{\text{DSR}}$ (Data Set Ready)—an inverting input used to test the $\overline{\text{DSR}}$ signal from the modem (data set). The $\overline{\text{DSR}}$ input indicates that the data set is ready to begin transferring information.

$\overline{\text{DTR}}$ (Data Terminal Ready)—an inverting output that signals the data set that the data terminal (8251A) is ready to transfer information.

RESET (Reset Input)—clears the internal circuitry of the 8251A, selecting the synchronous mode of operation. This input is connected to the RESET signal on the system bus.

$\overline{\text{RD}}$ (Read Input)—used to read data from the receiver or to read the status register.

$\overline{\text{RTS}}$ (Request-to-Send)—an inverting output that signals the data set that the 8251A is requesting that the line be turned around for transmission. This signal is used in the half-duplex mode of operation.

$\overline{\text{RxC}}$ (Receiver Clock)—provides a timing signal for the receiver.

RxD (Receiver Data)—accepts serial data for the receiver.

RxRDY (Receiver Ready)—shows that the receiver has received a serial datum that is ready for transfer in parallel to the microprocessor from the 8251A.

SY/BD (Sync Detect/Break Detect)—an output that indicates either synchronization in the synchronous modes or the receipt of a break character in the asynchronous mode. A break character is two complete frames of start pulses and is often used to break communications.

$\overline{\text{TxC}}$ (Transmitter Clock)—provides the 8251A transmitter with its Baud rate determining clock frequency. This input can be scaled by factors of 1, 16, or 64 in the asynchronous mode, and represents the actual Baud rate for synchronous operation.

TxD (Transmit Data)—serial data output connection.

TxEMPTY (Transmitter Empty)—indicates that the transmitter within the 8251A has finished transmitting all data.

TxRDY (Transmitter Ready)—a signal that indicates the transmitter within the 8251A is ready to receive another character for transmission.

$\overline{\text{WR}}$ (Write Input)—strobes data into the transmitter or the internal command register for programming.

## Programming the 8251A

Programming the 8251A is simple when compared to some of the other programmable interfaces described in this chapter. Programming is a two-part process that includes initialization dialog and operational dialog.

Initialization dialog, which occurs after a hardware or software reset, consists of two parts: reset and mode. Because of an apparent design flaw, the 8251A does not reset properly from the RESET input pin. Instead, it must be reset by a series of instructions that send the command register three 00Hs followed by a 40H (the software reset command).

Once the 8251A is reset, it may be programmed with the mode word, which directs the 8251A to function as either an asynchronous or synchronous device. Figure 11–43 shows both the synchronous and asynchronous mode command words. Both words specify the number of data bits and parity. In the asynchronous mode, the number of stop bits and clock divider are programmed; in the synchronous mode, the number of sync characters and the function of SY are programmed. The SY pin is programmed either as an input, which indicates synchronization, or as an output, which indicates the sync characters have been received.

In the asynchronous mode, the initialization programming is complete once the mode instruction is programmed. In the synchronous mode, the one or two sync characters are programmed following the mode command to complete initialization programming.

Suppose that an asynchronous system requires seven data bits, odd parity, a clock divider of 64, and one stop bit. Example 11–24 lists a procedure that initializes the 8251A to function in this manner. Figure 11–44 shows the interface to the 8086 microprocessor using a PAL16L8 to decode the 8-bit port addresses FDH and FFH. Here port FDH accesses the data register and port FFH accesses the command register. Notice that the Baud clock input is 76,800 Hz, and the transmitted and received serial data are at 1200 Baud. The reason is that the Baud rate multiplier is programmed to divide the input clock by a factor of 64 in this example.

**EXAMPLE 11–24**

```
                    ;Initialization procedure for the 8251A used in
                    ;asynchronous operation.
```

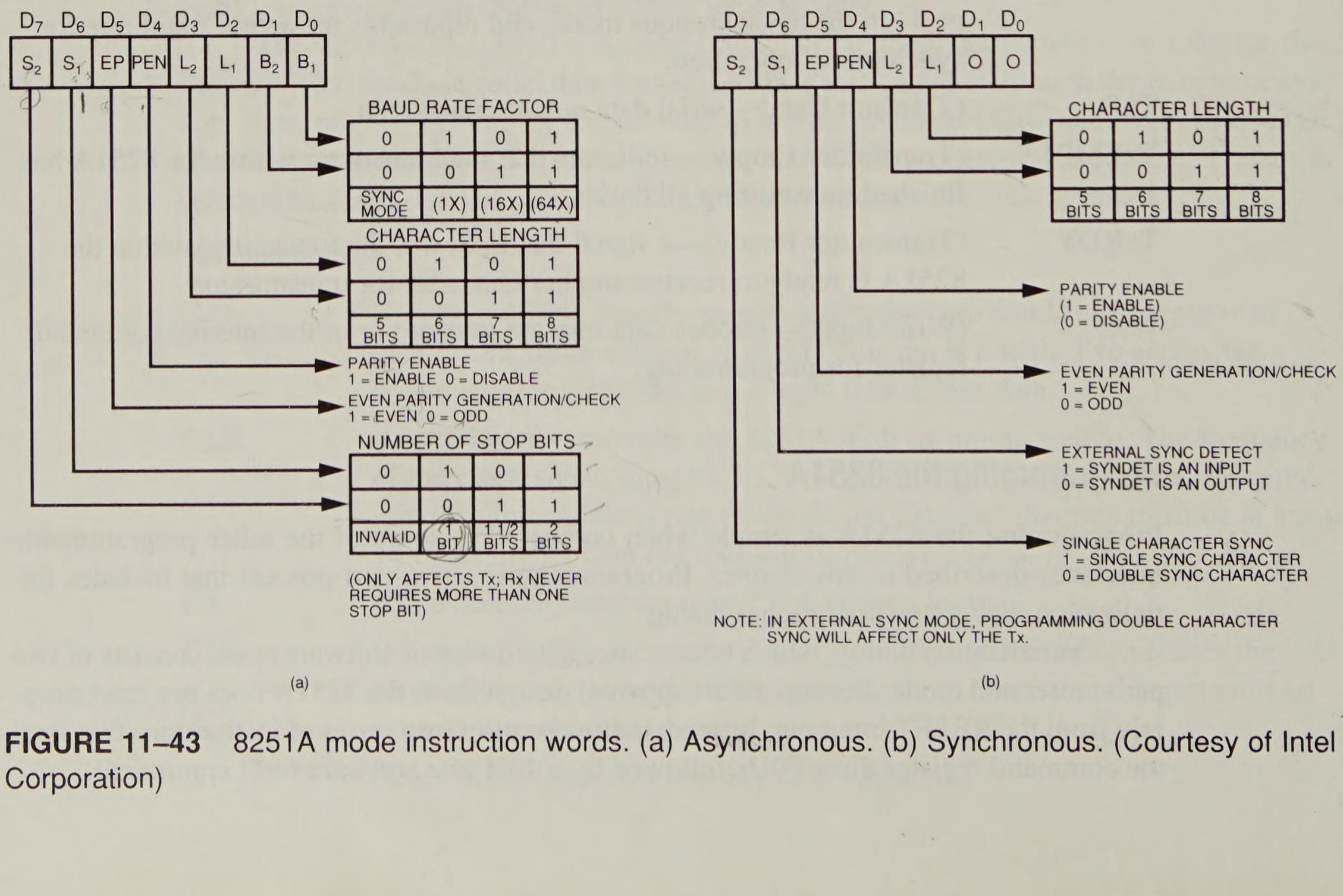



**FIGURE 11–43** 8251A mode instruction words. (a) Asynchronous. (b) Synchronous. (Courtesy of Intel Corporation)

```
                        ;
= 00FF                  CNTR    EQU  0FFH              ;command port
= 0040                  RESET   EQU  40H               ;reset code

0000                    PROG    PROC NEAR

000050                          PUSH AX                ;save AX

                        ;reset the 8251A

0001 32 C0                      XOR  AL,AL
0003 E6 FF                      OUT  CNTR,AL
0005 E6 FF                      OUT  CNTR,AL
0007 E6 FF                      OUT  CNTR,AL
0009 B0 40                      MOV  AL,RESET
000B E6 FF                      OUT  CNTR,AL

                        ;program mode

000D B0 5B                      MOV  AL,01011011B
000F E6 FF                      OUT  CNTR,AL

                        ;enable receiver and transmitter

0011 B0 15                      MOV  AL,00010101B
0013 E6 FF                      OUT  CNTR,AL

0015 58                         POP  AX                ;restore AX
0016 C3                         RET

0017                    PROG    ENDP
```

The 8251A programming sequence is very similar to the synchronous mode, except that one or two sync characters are also programmed. Example 11–25 lists a procedure that programs the 8251A for synchronous operation using seven data bits, even parity, SY as an output, and two sync characters.

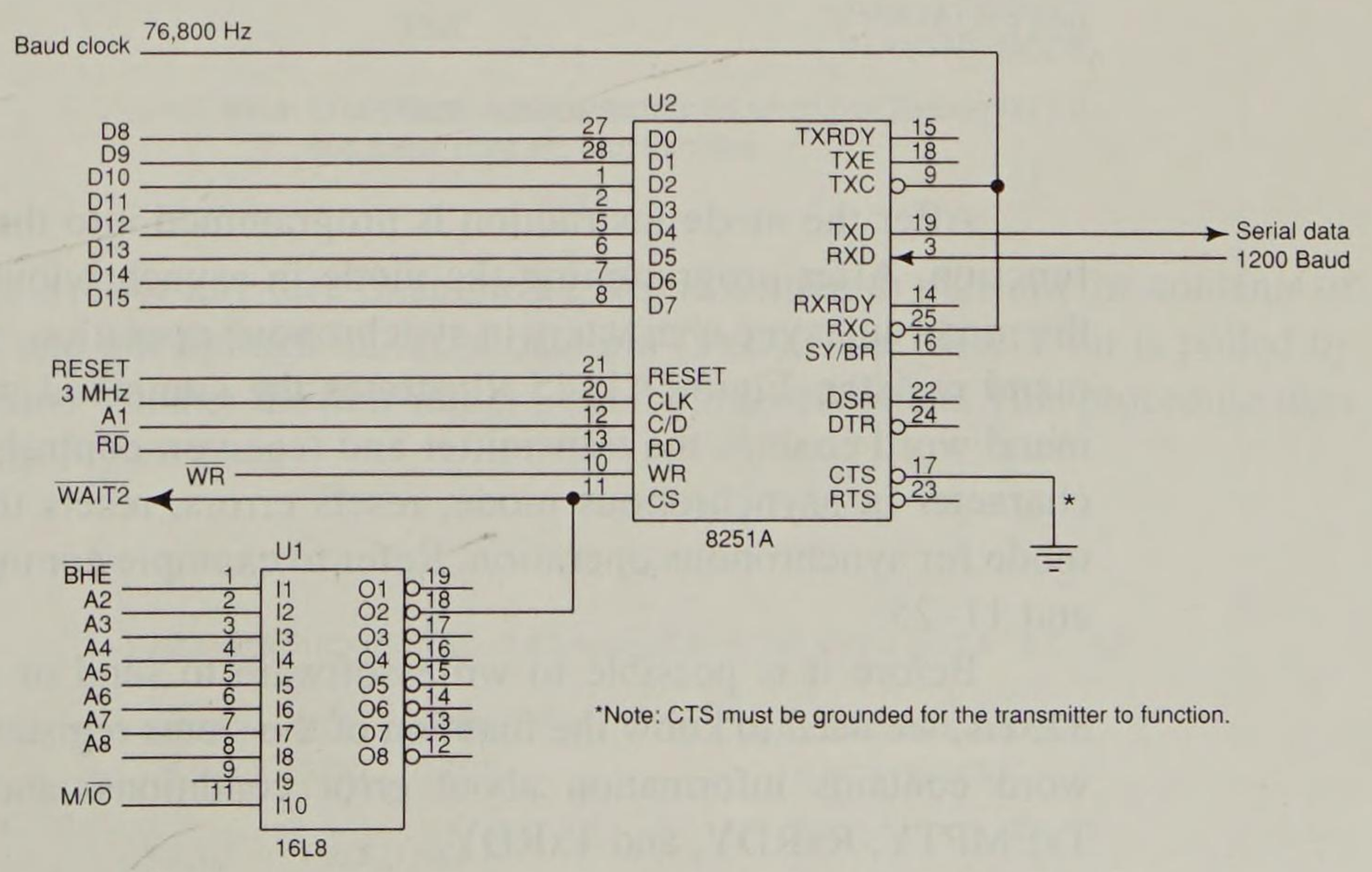



**FIGURE 11–44** The 8251A interfaced to the upper data bus of the 8086 at I/O ports FDH and FFH.

*Note: CTS must be grounded for the transmitter to function.

**EXAMPLE 11–25**

```
                        ;Initialization procedure for the 8251A used in
                        ;synchronous operation.
                        ;
= 00FF                  CNTR    EQU  0FFH              ;command port
= 0040                  RESET   EQU  40H               ;reset code
= 007F                  SYNC1   EQU  7FH               ;sync code 1
= 007E                  SYNC2   EQU  7EH               ;sync code 2

0000                    PROGS   PROC NEAR

0000 50                         PUSH AX                ;save AX

                        ;reset the 8251A

0001 32 C0                      XOR  AL,AL
0003 E6 FF                      OUT  CNTR,AL
0005 E6 FF                      OUT  CNTR,AL
0007 E6 FF                      OUT  CNTR,AL
0009 B0 40                      MOV  AL,RESET
000B E6 FF                      OUT  CNTR,AL

                        ;program mode

000D B0 B8                      MOV  AL,10111000B
000F E6 FF                      OUT  CNTR,AL

                        ;program sync characters

0011 B0 7F                      MOV  AL,SYNC1
0013 E6 FF                      OUT  CNTR,AL
0015 B0 7E                      MOV  AL,SYNC2
0017 E6 FF                      OUT  CNTR,AL

                        ;enable receiver and transmitter

0019 B0 15                      MOV  AL,00010101B
001B E6 FF                      OUT  CNTR,AL

001D 58                         POP  AX                ;restore AX
001E C3                         RET

001F                    PROGS   ENDP
```

After the mode instruction is programmed into the 8251A, it is still not ready to function. After programming the mode in asynchronous operation, and programming the mode and sync characters in synchronous operation, we still must program the command register. Figure 11–45 illustrates the command word for the 8251A. The command word enables the transmitter and receiver, controls DTR and RTS, sends a break character in asynchronous mode, resets errors, resets the 8251A, and enters the hunt mode for synchronous operation. Refer to example command words in Examples 11–24 and 11–25.

Before it is possible to write software to send or receive serial data through the 8251A, we need to know the function of the status register (see Figure 11–46). The status word contains information about error conditions, and the states of DSR, SY/BD, TxEMPTY, RxRDY, and TxRDY.

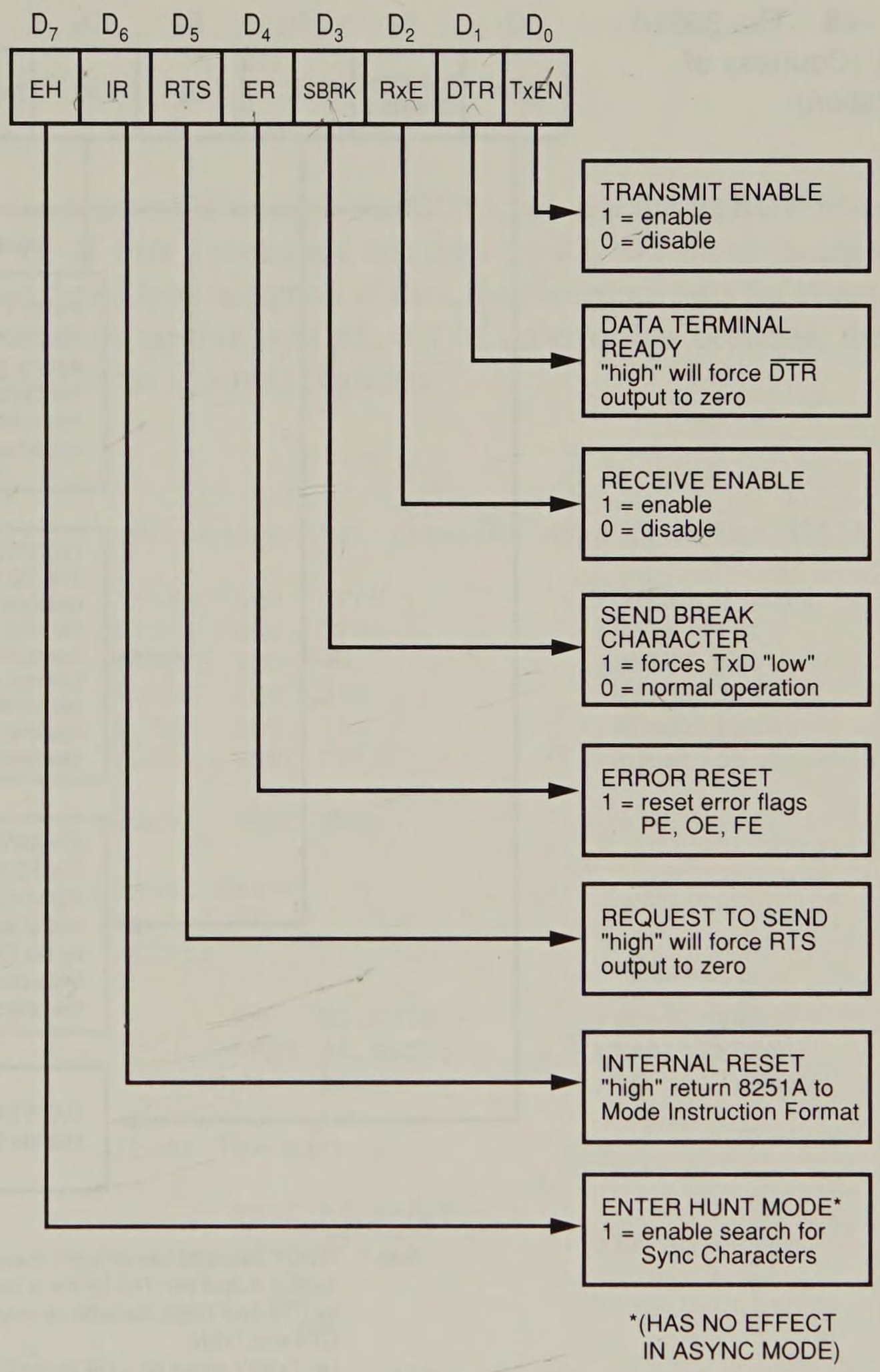



**FIGURE 11–45** The 8251A command word. (Courtesy of Intel Corporation)

**Note:** Error Reset must be performed whenever RxEnable and Enter Hunt are programmed.

Suppose that a procedure (see Example 11–26) is written to transmit the contents of AH to the 8251A and out through its serial data pin (TxD). The TxRDY bit is polled by software to determine whether the transmitter is ready to receive data. This procedure uses the circuit shown in Figure 11–43.

**EXAMPLE 11–26**

```
                        ;Procedure that transmits the contents of AH
                        ;
= 00FF                  CNTR    EQU  0FFH           ;command port
= 00FD                  DATA    EQU  0FDH           ;data port

0000                    SEND    PROC NEAR
```

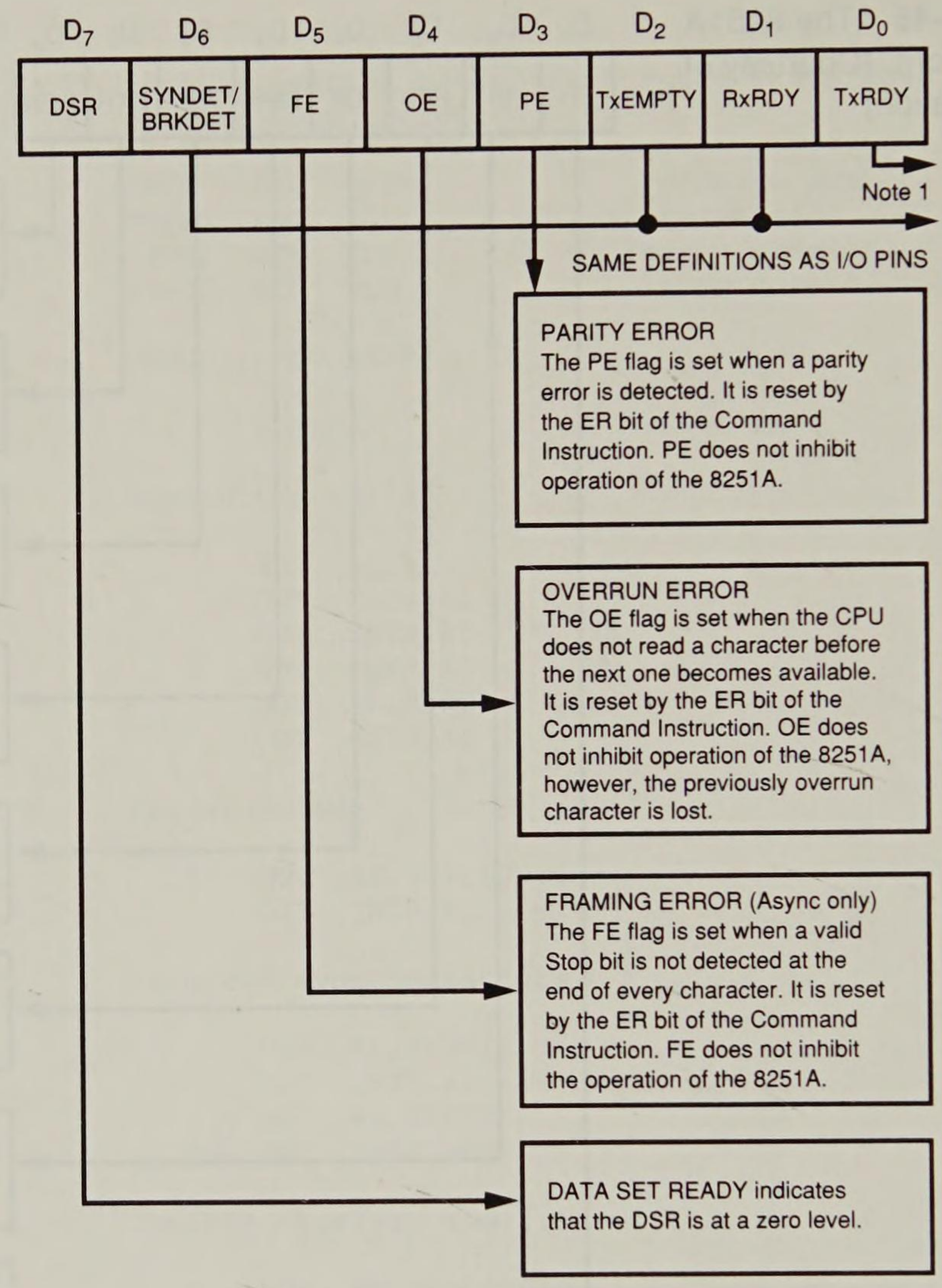



**FIGURE 11–46** The 8251A status word. (Courtesy of Intel Corporation)

Note 1: TxRDY status bit has different meanings from the TxRDY output pin: The former is not conditioned by $\overline{CTS}$ and TxEN; the latter is conditioned by both $\overline{CTS}$ and TxEN.
i.e. TxRDY status bit = DB Buffer Empty
TxRDY pin out = DB Buffer Empty •(CTS–0)• (TxEN-1)

```
0000 50                 PUSH AX             ;save AX

                        ;test TxRDY

0001                    SEND1:

0001 E4 FF                  IN   AL,CNTR    ;read status
0003 D0 C8                  ROR  AL,1       ;rotate TxRDY to CF
0005 73 FA                  JNC  SEND1      ;if not ready

                        ;transmit data

0007 8A C4                  MOV  AL,AH
0009 E6 FD                  OUT  DATA,AL
```

```
000B 58                         POP   AX              ;restore AX
000C C3                         RET

000D                    SEND    ENDP
```

To read received information from the 8251A, we test the RxRDY bit of the status register. Example 11–27 lists a procedure that tests the RxRDY bit to decide whether the 8251A has received data. Upon reception of data, the procedure tests for errors. If an error is detected, the procedure returns with AL = ?. If no error has occurred, the procedure returns with AL equal to the received character.

**EXAMPLE 11–27**

```
                        ;Procedure that receives data from the 8251A
                        ;
= 00FF                  CNTR    EQU   0FFH              ;command port
= 00FD                  DATA    EQU   0FDH              ;data port
= 0004                  MASKS   EQU   4                 ;RxRDY mask
= 0038                  ERROR   EQU   38H               ;error mask
= 0015                  E_RES   EQU   15H               ;error reset
= 003F                  QUES    EQU   '?'               ;question mark

0000                    RECV    PROC NEAR

                        ;test RxRDY

0000                    RECV1:

0000 E4 FF                      IN    AL,CNTR           ;read status
0002 A8 04                      TEST  AL,MASKS          ;test RxRDY
0004 74 FA                      JZ    RECV1             ;if not ready

                        ;test for errors

0006 A8 38                      TEST  AL,ERROR
0008 75 03                      JNZ   ERR               ;if an error

                        ;read data

000A E4 FD                      IN    AL,DATA
000C C3                         RET

                        ;error

000D                    ERR:

000D B0 15                      MOV   AL,E_RES          ;reset error
000F E6 FF                      OUT   CNTR,AL
0011 B0 3F                      MOV   AL,QUES           ;get ?
0013 C3                         RET

0014                    RECV    ENDP
```

The types of errors detected by the 8251A are parity error, framing error, and overrun error. A parity error indicates that the received data contain the wrong parity. A framing error indicates that the start and stop bits are not in their proper places. An overrun

error indicates that data has overrun the internal receiver buffer. These errors should not occur during normal operation. If a parity error occurs, it indicates that noise was encountered during reception. A framing error occurs if the receiver is receiving data at an incorrect Baud rate. An overrun error only occurs if the software fails to read the data from the USART.

## 11–7 ANALOG-TO-DIGITAL (ADC) AND DIGITAL-TO-ANALOG (DAC) CONVERTERS

Analog-to digital (ADC) and digital-to-analog (DAC) converters are used to interface the microprocessor to the analog world. Many events that are monitored and controlled by the microprocessor are analog events. All sorts of events—such as speech and the control of motors—are analogs. In order to interface the microprocessor to these events, we must have an understanding of the interface and control of the ADC and DAC, which convert between analog and digital data.

### The DAC0830 Digital-to-Analog Converter

One common and inexpensive digital-to-analog converter is the DAC0830.* This device is an 8-bit converter that transforms an 8-bit binary number into an analog voltage. Other converters that convert 10, 12, or 16 bits into analog voltages are available. The number of voltage steps generated by the converter is equal to the number of binary input combinations. Therefore, an 8-bit converter generates 256 different voltage levels, a 10-bit converter generates 1,024 levels, and so forth. The DAC0830 is a medium-speed converter that transforms a digital input to an analog output in approximately 1.0 μs.

Figure 11–47 illustrates the pinout of the DAC0830. This device has a set of eight data bus connections for the application of the digital input code and a pair of analog outputs labeled Iout1 and Iout2 that are designed as inputs to an external operational amplifier. Because this is an 8-bit converter, its output step voltage is defined as –Vref (reference voltage) divided by 255. For example, if the reference voltage is –5.0 V, its output step voltage is +.0196 V. Note that the output voltage is the opposite polarity of the reference voltage. If an input of 1001 0010 is applied to the device, the output voltage will be the step voltage times 1001 0010, or, in this case, +2.862 V. By changing the reference voltage to –5.1 V, the step voltage becomes +.02 V. The step voltage is often called the resolution of the converter.

*Internal Structure of the DAC0830.* Figure 11–48 illustrates the internal structure of the DAC0830. Notice that this device contains 2 internal registers. The first is a holding register, while the second connects to the R–2R internal ladder converter. The two latches allow one byte to be held while another is converted. In many cases we disable the first latch and use only the second for entering data into the converter. This is accomplished by connecting a logic 1 to ILE and a logic 0 to $\overline{CS}$ (chip select).

*The DAC0830 is a product of National Semiconductor Corporation.

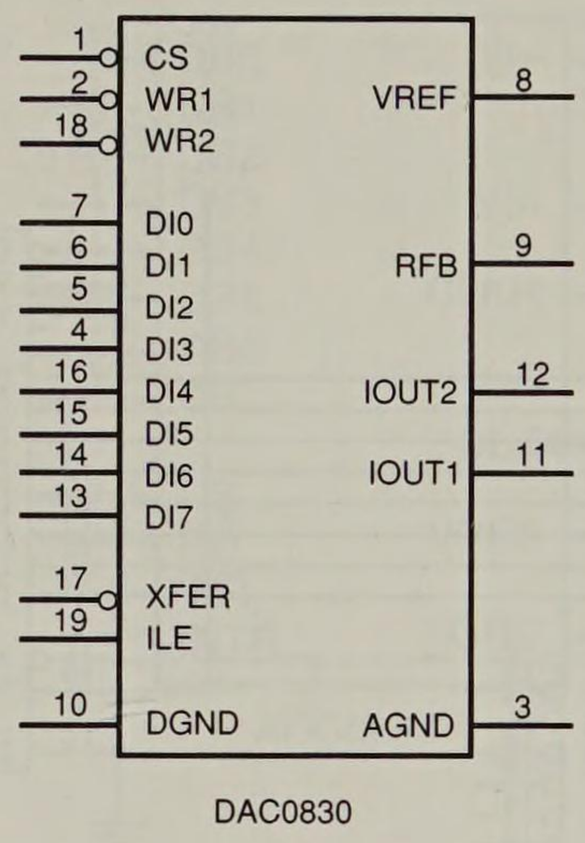



**FIGURE 11–47** The pinout of the DAC0830 digital-to-analog converter.

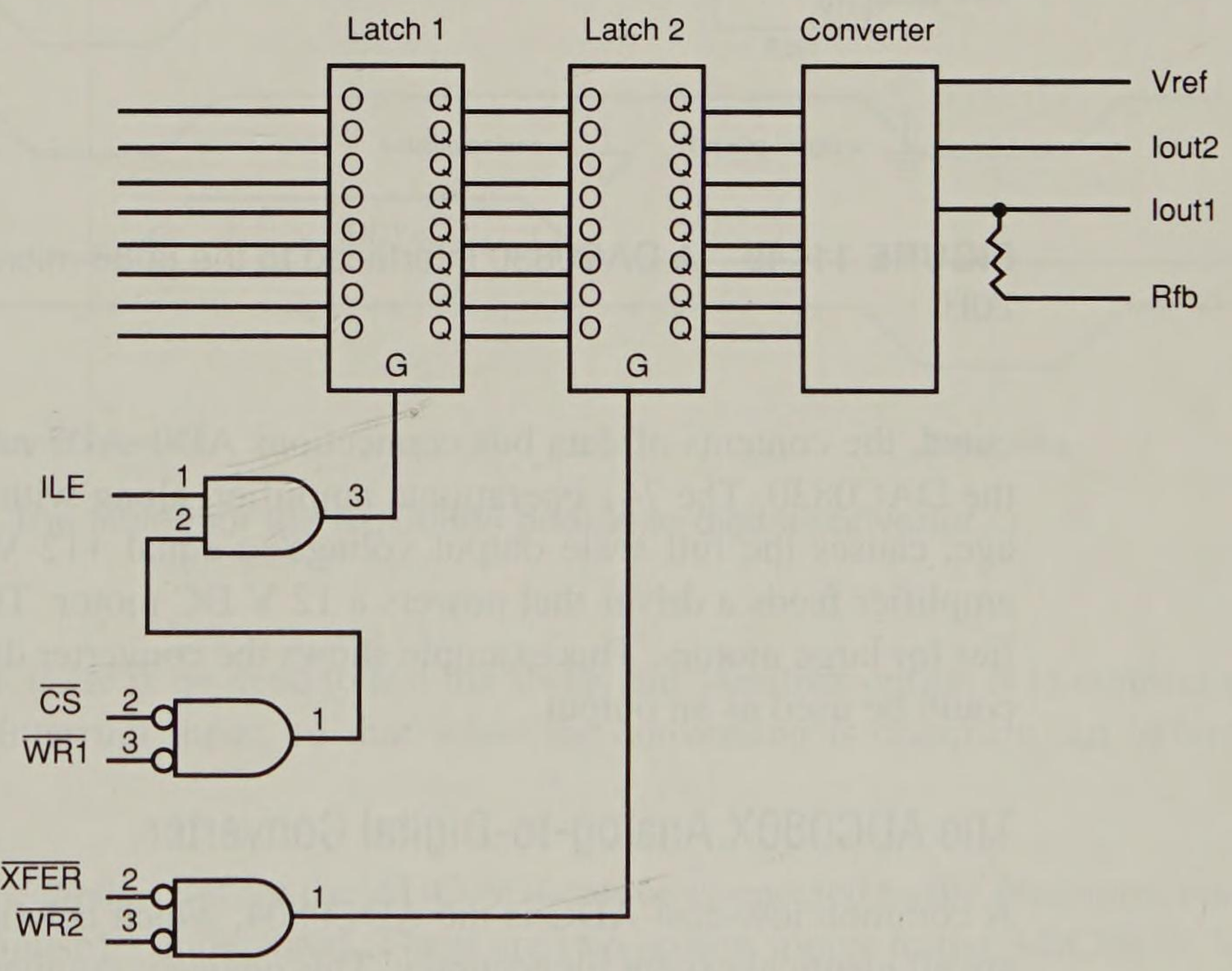



**FIGURE 11–48** The internal structure of the DAC0830.

Both latches within the DAC0830 are transparent latches. When the G input to the latch is a logic 1, data pass through the latch; when the G input to the latch becomes a logic 0, data are latched or held. The converter has a reference input pin (Vref) that establishes the full-scale output voltage. If –10 V is placed on Vref, the full-scale (1111 1111) output voltage is +10 V. The output of the R–2R ladder within the converter appears at Iout1 and Iout2. These outputs are designed to be applied to an operational amplifier, such as a 741 or similar device.

*Connecting the DAC0830 to the Microprocessor.* The DAC0830 is connected to the microprocessor as illustrated in Figure 11–49. Here a PAL16L8 is used to decode the DAC0830 at 8-bit I/O port address 20H. Whenever an OUT 20H,AL instruction is exe-

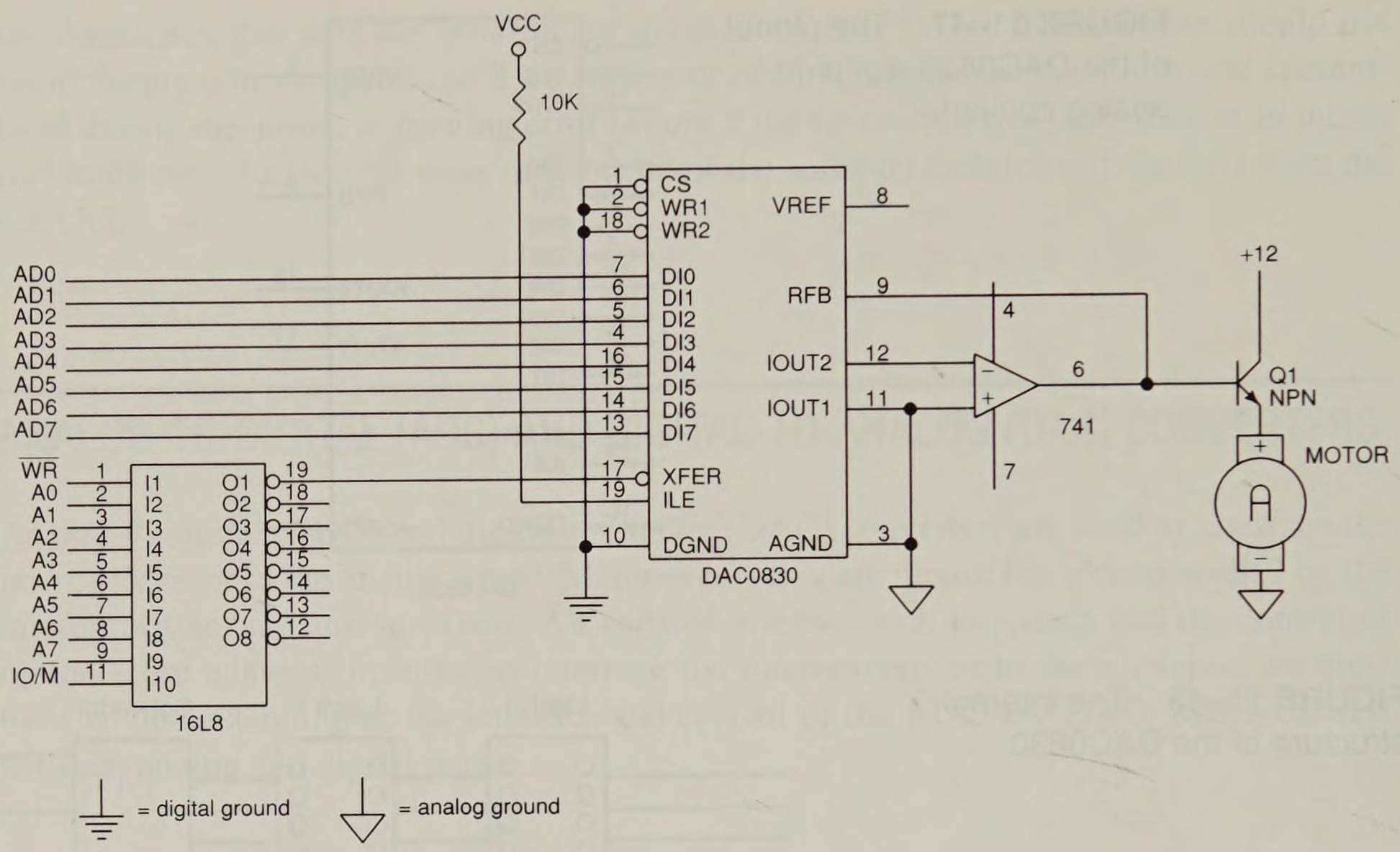



**FIGURE 11–49** A DAC0830 interfaced to the 8086 microprocessor at 8-bit I/O location 20H.

cuted, the contents of data bus connections AD0–AD7 are passed to the converter within the DAC0830. The 741 operational amplifier, along with the –12 V zener reference voltage, causes the full scale output voltage to equal +12 V. The output of the operational amplifier feeds a driver that powers a 12 V DC motor. This driver is a Darlington amplifier for large motors. This example shows the converter driving a motor, but other devices could be used as an output.

## The ADC080X Analog-to-Digital Converter

A common low-cost ADC is the ADC0804, which belongs to a family of converters that are all identical except for accuracy. This device is compatible with a wide range of microprocessors, such as the 8086. There are faster ADCs available, and some with more resolution than 8 bits, but this device is ideal for applications that do not require a high degree of accuracy. The ADC0804 requires up to 100 μs to convert an analog input voltage into a digital output code.

Figure 11–50 shows the pinout of the ADC0804* converter. The $\overline{WR}$ pin is pulsed with $\overline{CS}$ grounded to start the conversion process. Because this converter requires a considerable amount of time for the conversion, a pin labeled INTR signals the end of the conversion. Refer to Figure 11–51 for a timing diagram that shows the interaction of the control signals. Start the converter with the $\overline{WR}$ pulse, wait for INTR to return to a logic 0 level, and then read the data from the converter. If a time delay that allows at least 100

*The ADC0804 is a product of National Semiconductor Corporation.

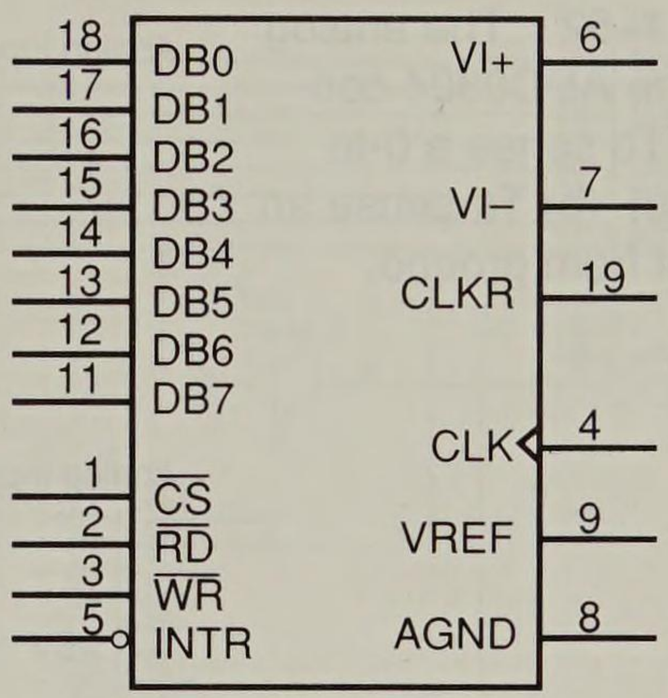



**FIGURE 11–50** The pinout of the ADC0804 analog-to-digital converter.

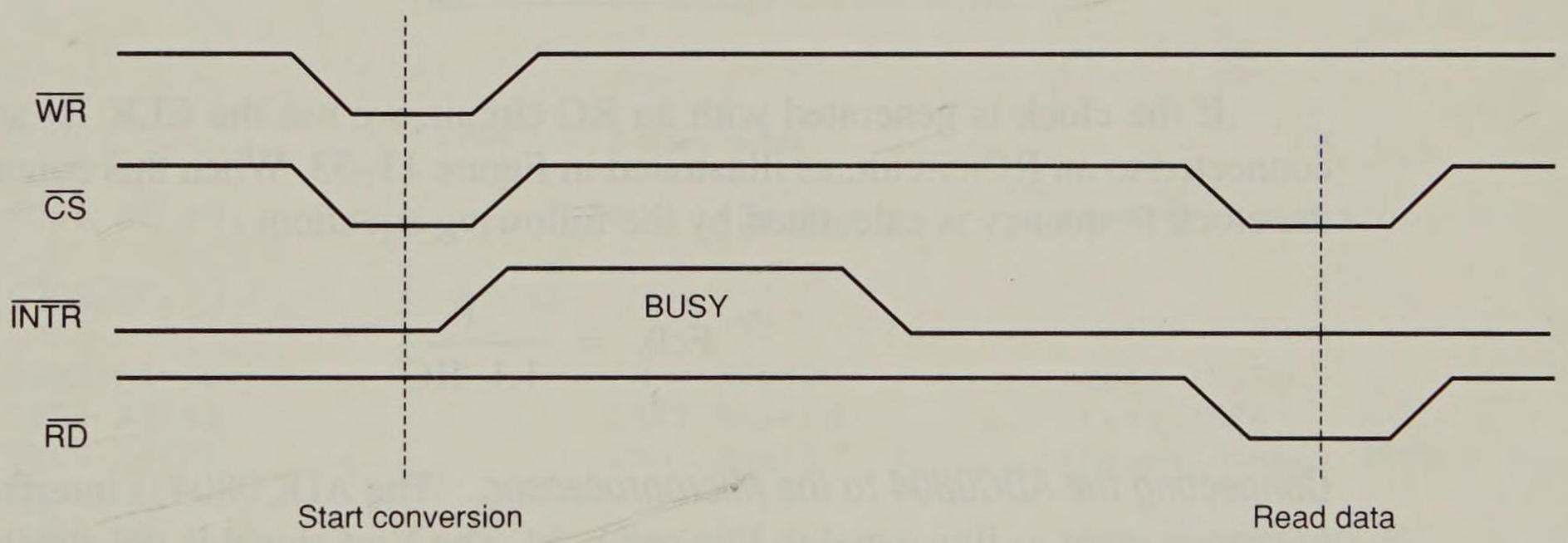



**FIGURE 11–51** The timing for the ADC0804 analog-to-digital converter.

μs of time is used, there is no need to test the INTR pin. Another option is to connect the INTR pin to an interrupt input, so that when the conversion is complete, an interrupt occurs.

*The Analog Input Signal.* Before the ADC0804 can be connected to the microprocessor, its analog inputs must be understood. There are two analog inputs to the ADC0804: Vin (+) and Vin (–). These inputs are connected to an internal operational amplifier and are differential inputs as shown in Figure 11–52. The differential inputs are summed by the operational amplifier to produce a signal for the internal analog-to-digital converter. Figure 11–52 shows a few ways to use these differential inputs. The first way (Figure 11–52a) uses a single input that can vary between 0 V and +5.0 V. The second way (Figure 11–52b) uses a variable voltage applied to the Vin (–) pin so the zero reference for VIN (+) can be adjusted.

*Generating the Clock Signal.* The ADC0804 requires a clock source for operation. The clock can be an external clock applied to the CLK IN pin, or it can be generated with an RC circuit. The permissible range of clock frequencies is between 100 KHz and 1460 KHz. Use a frequency as close as possible to 1460 KHz, so conversion time is kept to a minimum.

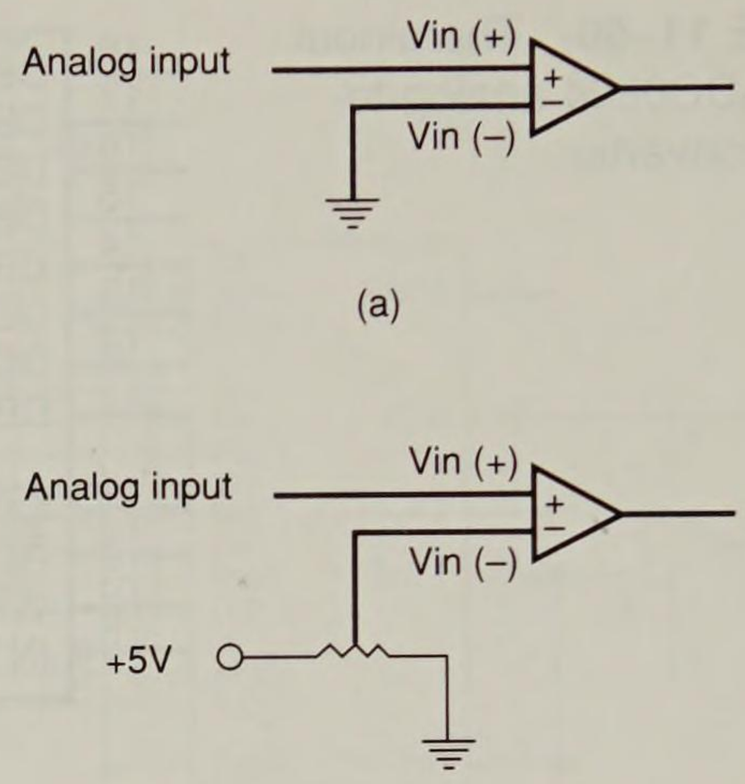



**FIGURE 11–52** The analog inputs to the ADC0804 converter. (a) To sense a 0-to +5.0-V input. (b) To sense an input offset from ground.

If the clock is generated with an RC circuit, we use the CLK IN and CLK R pins connected to an RC circuit, as illustrated in Figure 11–53. When this connection is in use, the clock frequency is calculated by the following equation:

$$Fclk = \frac{1}{1.1\ RC}$$

*Connecting the ADC0804 to the Microprocessor.* The ADC0804 is interfaced to the 8086 microprocessor as illustrated in Figure 11–54. The Vref signal is not attached to anything, and this is normal. Suppose that the ADC0804 is decoded at 8-bit I/O port address 40H for the data and port address 42H for the INTR signal, and a procedure is required to start and read the data from the ADC. This procedure is listed in Example 11–28. Notice that the INTR bit is polled: if it becomes a logic 0, the procedure ends with AL containing the converted digital code.

**EXAMPLE 11–28**

```
;Procedure that reads data from the ADC and returns
;with it in AL
;
```

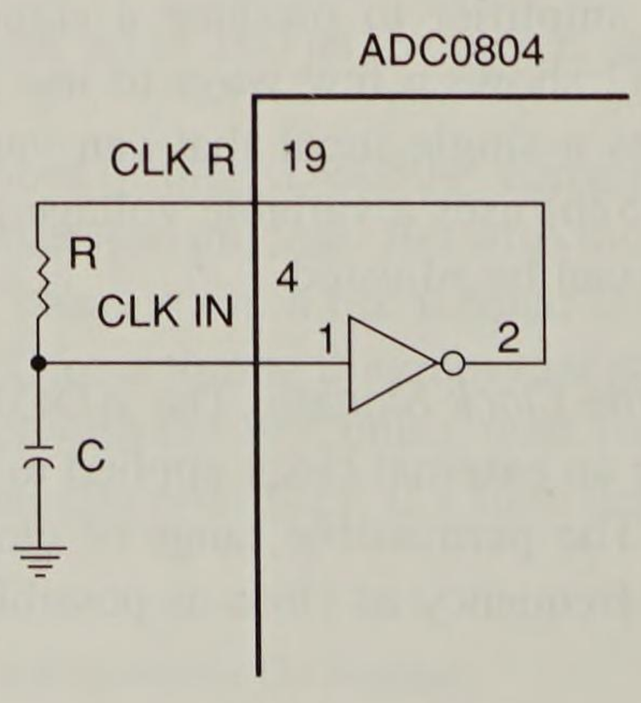
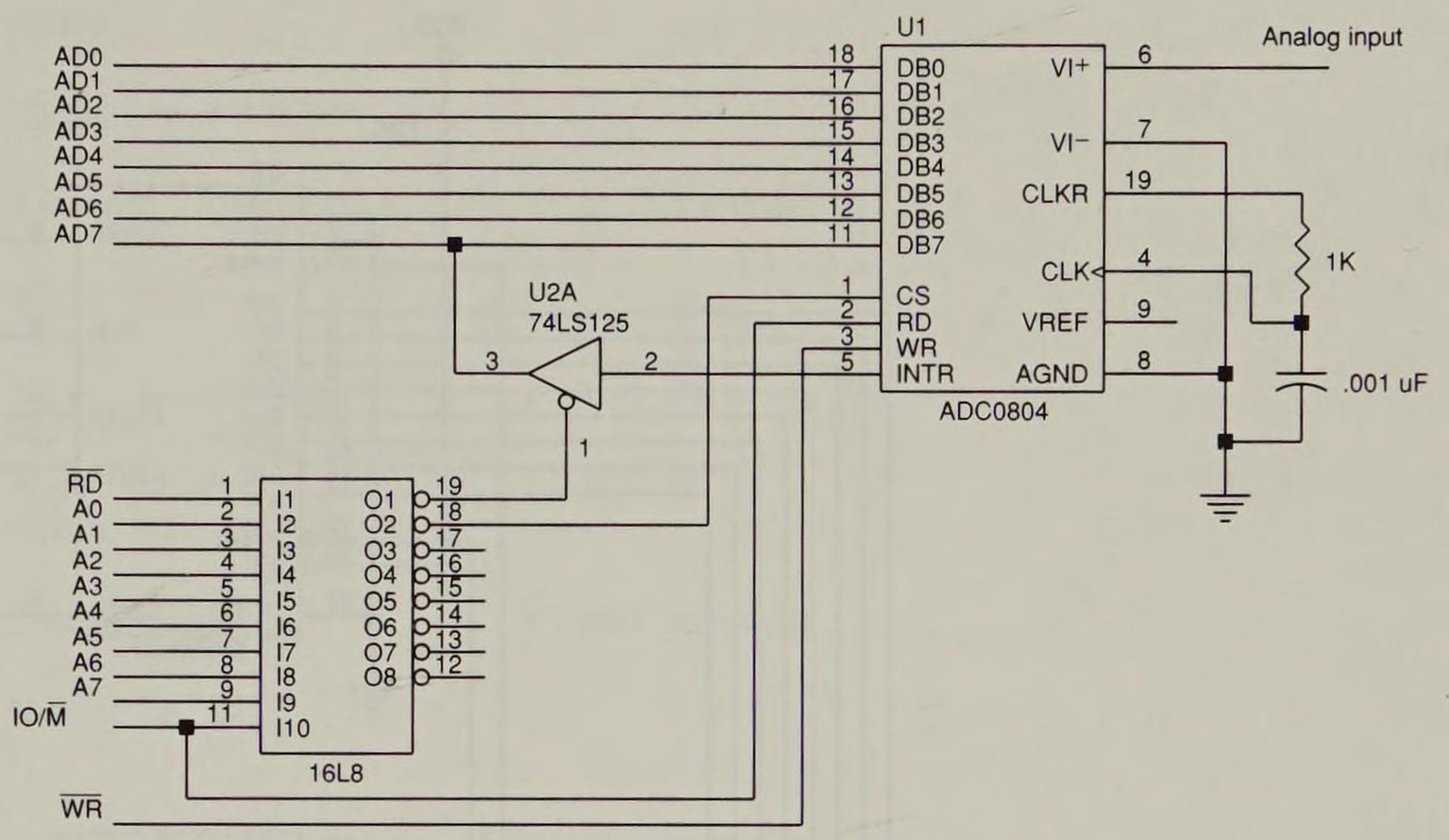



**FIGURE 11–53** Connecting the RC circuit to the CLK IN and CLK R pins on the ADC0804.




```
0000                    ADCX    PROC NEAR

0000 E6 40                      OUT  40H,AL          ;start conversion

0002                    ADCX1:

0002 E4 42                      IN   AL,42H          ;read INTR
0004 A8 80                      TEST AL,80H          ;test INTR
0006 75 FA                      JNZ  ADCX1           ;repeat until INTR = 0

0008 E4 40                      IN   AL,40H          ;get data from ADC
000A C3                         RET

000B                    ADCX    ENDP
```

## Using the ADC0804 and the DAC0830

This section of the text illustrates an example using both the ADC0804 and the DAC0830 to capture and replay audio signals or speech. In the past we often used a speech synthesizer to generate speech, but the quality of the speech was poor. For human quality speech, we can use the ADC0804 to capture an audio signal and store it in memory for later playback through the DAC0830.

Figure 11–55 illustrates the circuitry required to connect the ADC0804 at 16-bit I/O ports 0700H and 0702H. The DAC0830 is interfaced at I/O port 704H. The software used to run these converters appears in Example 11–29. This software reads a one-second burst of speech, and then plays it back 10 times. This process repeats until the system is turned off.

**EXAMPLE 11–29** (pp. 511–514)

```
                        ;Software that records a 1 second passage of speech and
                        ;plays it back 10 times before recording the next 1 second
                        ;of speech.
                        ;
                        ;Assumes that the clock is 8 MHz
                        ;
0000                    STAC    SEGMENT STACK
```

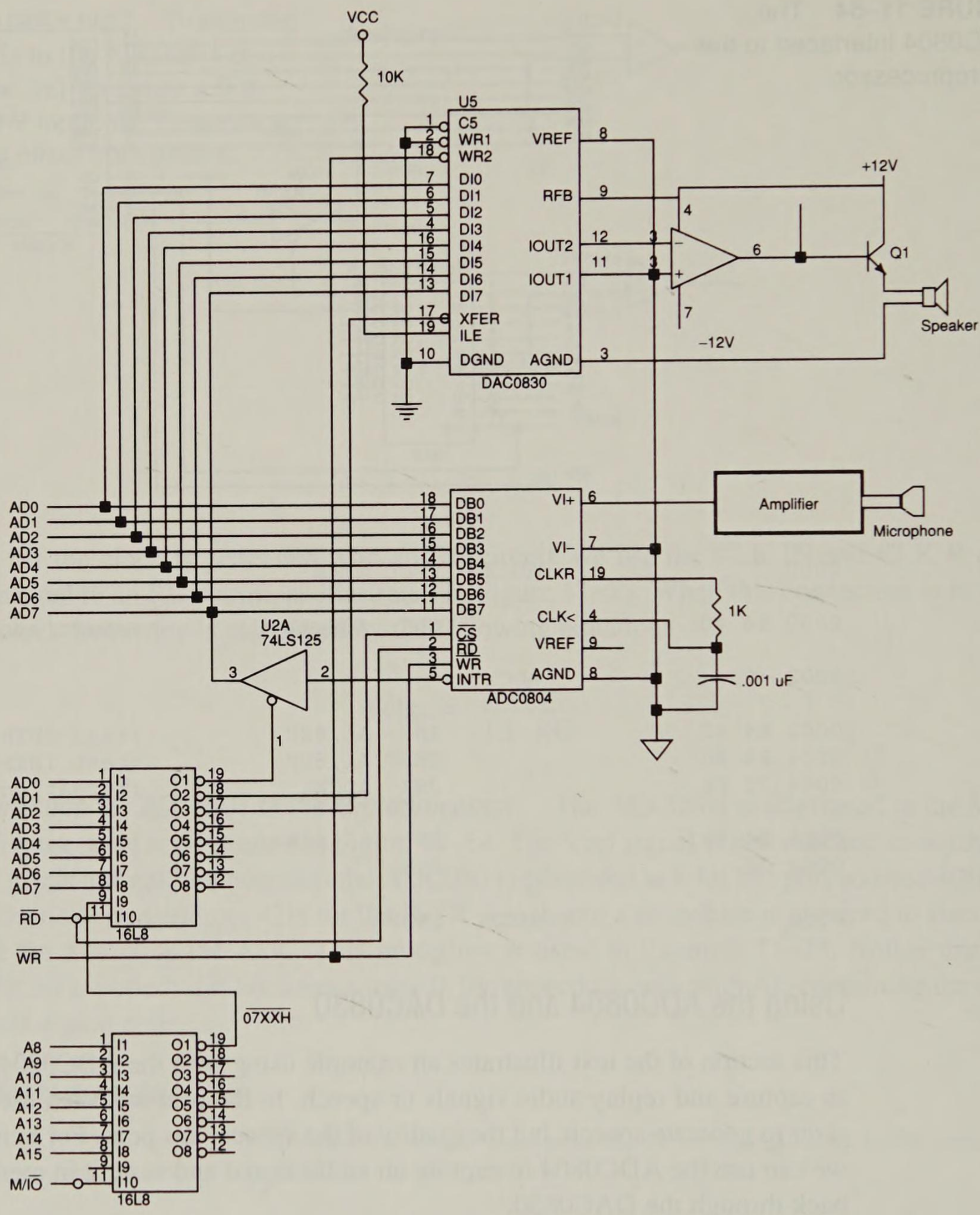



**FIGURE 11–55** A circuit to store speech and play it back through a loudspeaker.

```
0000 0800 [                DW   2048 DUP (?)     ;stack space
           0000
                ]

1000                  STAC   ENDS

0000                  DATA1  SEGMENT 'DATA'

0000 0400 [           WORDS  DB   1024 DUP (?)     ;space for 1 second of speech
           00
              ]
```

```
03E8                    DATA1   ENDS

0000                    CODE1   SEGMENT 'CODE'

                                ASSUME  CS:CODE1,DS:DATA1,SS:STAC

0000                    START   PROC FAR

0000 B8 ---- R                  MOV   AX,DATA1          ;load DS
0003 8E D8                      MOV   DS,AX
0005 33 C0                      XOR   AX,AX             ;load ES
0007 8E C0                      MOV   ES,AX

0009 E8 000A                    CALL READ               ;read speech

000C B9 000A                    MOV   CX,10

000F                    START1:

000F E8 0023                    CALL WRITE              ;talk
0012 E2 FB                      LOOP START1             ;repeat 10 times

0014 EB EA                      JMP   START             ;redo all

0016                    START   ENDP

0016                    READ    PROC NEAR

0016 BF 0000 R                  MOV   DI,OFFSET WORDS ;address data
0019 B9 03E8                    MOV   CX,1000           ;load count
001C BA 0700                    MOV   DX,0700H          ;address port

001F                    READ1:

001F EE                         OUT   DX,AL             ;start converter
0020 83 C2 02                   ADD   DX,2

0023                    READ2:

0023 EC                         IN    AL,DX             ;get INTR
0024 A8 80                      TEST  AL,80H            ;test INTR
0026 75 FB                      JNZ   READ2             ;wait for INTR = 0

0028 83 EA 02                   SUB   DX,2
002B EC                         IN    AL,DX             ;get data
002C 88 05                      MOV   [DI],AL           ;save data
002E 47                         INC   DI

002F E8 0018                    CALL DELAY              ;wait 1/1000 second

0032 E2 EB                      LOOP READ1              ;repeat 1,024 times
0034 C3                         RET

0035                    READ    ENDP

0035                    WRITE   PROC NEAR

0035 51                         PUSH CX
0036 BF 0000 R                  MOV   DI,OFFSET WORDS ;address data
0039 B9 03E8                    MOV   CX,1000           ;load count
003C BA 0704                    MOV   DX,0704H          ;address port
```

```
003F                    WRITE1:

003F 8A 05                      MOV  AL,[DI]        ;get data
0041 EE                         OUT  DX,AL          ;output data
0042 47                         INC  DI

0043 E8 0004                    CALL DELAY          ;wait 1/1000 second

0046 E2 F7                      LOOP WRITE1
0048 59                         POP  CX
0049 C3                         RET

004A                    WRITE  ENDP

004A                    DELAY  PROC NEAR

                        ;wait 1/1000 second

004A 51                         PUSH CX
004B B9 03E8                    MOV  CX,1000

004E                    DELAY1:

004E E2 FE                      LOOP DELAY1
0050 59                         POP  CX
0051 C3                         RET

0052                    DELAY  ENDP

0052                    CODE1  ENDS

                                END  START
```

## 11–8 SUMMARY

1. The 8086–Pentium has two basic types of I/O instructions: IN and OUT. The IN instruction inputs data from an external I/O device into either the AL (8-bit) or AX (16-bit) register. The IN instruction is available as a fixed-port instruction, a variable-port instruction, or a string instruction (INSB or INSW for the 80286–Pentium). The OUT instruction outputs data from AL or AX to an external I/O device and is available as a fixed, variable, or string instruction, OUTSB or OUTSW. The fixed-port instruction uses an 8-bit I/O port address, while the variable and string I/O instructions use a 16-bit port number found in the DX register.
2. Isolated I/O, sometimes called direct I/O, uses a separate map for the I/O space, freeing the entire memory for use by the program. Isolated I/O uses the IN and OUT instructions to transfer data between the I/O device and the microprocessor. The control structure of the I/O map uses $\overline{\text{IORC}}$ (I/O read control) and $\overline{\text{IOWC}}$ (I/O write control), plus the bank selection signals $\overline{\text{BHE}}$ and A0 to effect the I/O transfer. The early 8086/8088 used the M/$\overline{\text{IO}}$ (IO/$\overline{\text{M}}$) signal with RD and WR to generate the I/O control signals.
3. Memory-mapped I/O uses a portion of the memory space for I/O transfers. This reduces the amount of memory available, but negates the need to use the $\overline{\text{IORC}}$ and

$\overline{\text{IOWC}}$ signals for I/O transfers. Any instruction that addresses a memory location using any addressing mode can be used to transfer data between the microprocessor and the I/O device using memory-mapped I/O.

4. All input devices are buffered, so the I/O data are only connected to the data bus during the execution of the IN instruction. The buffer is either built into a programmable peripheral or located separately.
5. All output devices use a latch to capture output data during the execution of the OUT instruction. This is necessary because data appear on the data bus for less than 100 ns for an OUT instruction, and most output devices require the data for a longer time. In many cases the latch is built into the peripheral.
6. Handshaking or polling is the act of two independent devices synchronizing with a few control lines. For example, the computer asks a printer if it is busy by inputting the BUSY signal from the printer. If it isn't busy, the computer outputs data to the printer and informs the printer that data are available with a data strobe ($\overline{\text{DS}}$) signal. This communication between the computer and the printer is a handshake or a poll.
7. The I/O port number appears on address bus connections A7–A0 for a fixed port I/O instruction, and on address bus connections A15–A0 for a variable port I/O instruction. In both cases, address bits above A15 are zero. The 8-bit I/O address found on A7–A0 also contains logic 0s on address connections A15–A8.
8. Because the 8086/80286/80386SX contain a 16-bit data bus, and the I/O addresses reference byte-sized I/O locations, the I/O space is organized in banks like the memory system. In order to interface an 8-bit I/O device to the 16-bit data bus, we often require separate upper and lower write strobes for I/O write operations.
9. The I/O port decoder is much like the memory address decoder, except instead of decoding the entire address, the I/O port decoder decodes only a 16-bit address for variable port instructions, or an 8-bit port number for fixed I/O instructions.
10. The 8255 is a programmable peripheral interface (PIA) that has 24 I/O pins that are programmable in groups of 12 pins (group A and group B). The 8255 operates in three modes: simple I/O (mode 0), strobed I/O (mode 1), and bidirectional I/O (mode 2). When the 8255 is interfaced to the 8086 operating at 8 MHz, we insert two wait states, because the speed of the microprocessor is faster than the 8255 can handle.
11. The 8279 is a programmable keyboard/display controller that can control a 64-key keyboard and a 16-digit numeric display.
12. The 8254 is a programmable interval timer that contains three 16-bit counters that count in binary or binary-coded decimal (BCD). Each counter is independent of the other counters, and operates in six different modes. The six modes of the counter are (1) events counter, (2) retriggerable monostable multivibrator, (3) pulse generator, (4) squarewave generator, (5) software-triggered pulse generator, and (6) hardware-triggered pulse generator.
13. The 8251A is a programmable communications interface capable of receiving and transmitting asynchronous or synchronous serial data.
14. The DAC0830 is an 8-bit digital-to-analog converter that converts a digital signal to an analog voltage within 1 μs.
15. The ADC0804 is an 8-bit analog-to-digital converter that converts an analog signal into a digital signal within 100 μs.

## 11–9 QUESTIONS AND PROBLEMS

1. Explain which way the data flow for an IN and an OUT instruction.
2. Where is the I/O port number stored for a fixed I/O instruction?
3. Where is the I/O port number stored for a variable I/O instruction?
4. Where is the I/O port number stored for a string I/O instruction?
5. To which register are data input by the 16-bit IN instruction?
6. Describe the operation of the OUTSB instruction.
7. Describe the operation of the INSW instruction.
8. Contrast a memory-mapped I/O system with an isolated I/O system.
9. What is the basic input interface?
10. What is the basic output interface?
11. Explain the term *handshaking* as it applies to computer I/O systems.
12. An even-number I/O port address is found in the _____ I/O bank in the 8086 microprocessor.
13. Show the circuitry required to generate the upper and lower I/O write strobes.
14. Develop an I/O port decoder, using a 74ALS138, that generates low-bank I/O strobes for the 8-bit I/O port addresses 10H, 12H, 14H, 16H, 18H, 1AH, 1CH, and 1EH.
15. Develop an I/O port decoder, using a 74ALS138, that generates high-bank I/O strobes for the 8-bit I/O port addresses 11H, 13H, 15H, 17H, 19H, 1BH, 1DH, and 1FH.
16. Develop an I/O port decoder, using a PAL16L8, that generates 16-bit I/O strobes for the 16-bit I/O port addresses 1000H–1001H, 1002H–1003H, 1004H–1005H, 1006H–1007H, 1008H–1009H, 100AH–100BH, 100CH–100DH, and 100EH–100FH.
17. Develop an I/O port decoder, using the PAL16L8, that generates low-bank I/O strobes 00A8H, 00B6H, and 00EEH.
18. Develop an I/O port decoder, using the PAL16L8, that generates the following high-bank I/O strobes: 300DH, 300BH, 1005H, and 1007H.
19. Why are both $\overline{\text{BHE}}$ and A0 ignored in a 16-bit port address decoder?
20. An 8-bit I/O device, located at I/O port address 0010H, is connected to which data bus connections?
21. An 8-bit I/O device located at I/O port address 100DH is connected to which data bus connections?
22. The 8255 has how many programmable I/O pin connections?
23. List the pins that belong to group A and to group B in the 8255.
24. What two 8255 pins accomplish internal I/O port address selection?
25. The $\overline{\text{RD}}$ connection on the 8255 is attached to which 8086 system control bus connection?
26. Using a PAL16L8, interface an 8255 to the 8086 microprocessor, so it functions at I/O locations 0380H, 0382H, 0384H, and 0386H.
27. When the 8255 is reset, its I/O ports are all initialized as _____.
28. What three modes of operation are available to the 8255?
29. What is the purpose of the $\overline{\text{STB}}$ signal in strobed input operation of the 8255?
30. Explain the operation of a simple four-coil stepper motor.
31. What sets the IBF pin in strobed input operation of the 8255?
32. Write the software required to place a logic 1 on the PC7 pin of the 8255 during strobed input operation.

33. How is the interrupt request pin (INTR) enabled in the strobed input mode of operation of the 8255?
34. In strobed output operation of the 8255, what is the purpose of the $\overline{ACK}$ signal?
35. What clears the $\overline{OBF}$ signal in strobed output operation of the 8255?
36. Write the software required to decide whether PC4 is a logic 1, when the 8255 is operated in the strobed output mode.
37. Which group of pins is used during bidirectional operation of the 8255?
38. What pins are general-purpose I/O pins during mode 2 operation of the 8255?
39. What is normally connected to the CLK pin of the 8279?
40. How many wait states are required to interface the 8279 to the 8086 microprocessor operating with an 8 MHz clock?
41. If the 8279 CLK pin is connected to a 3.0 MHz clock, program the internal clock.
42. What is an overrun error in the 8279?
43. What is the difference between *encoded* and *decoded*, as defined for the 8279?
44. Interface the 8279 so it functions at 8-bit I/O ports 40H–7FH. Use the 74ALS138 as a decoder and use either the upper or lower data bus.
45. Interface a 16-key keyboard and an 8-digit numeric display to the 8279.
46. The 8254 interval timer functions from DC to _____ Hz.
47. Each counter in the 8254 functions in how many different modes?
48. Interface an 8254 to function at I/O port addresses XX10H, XX12H, XX14H, and XX16H. Write the software required to cause counter 2 to generate an 80 KHz squarewave if the CLK input to counter 2 is 8 MHz.
49. What number is programmed in an 8254 counter to count 300 events?
50. If a 16-bit count is programmed into the 8254, which byte of the count is programmed first?
51. Explain how the read-back control word functions in the 8254.
52. Program counter 1 of the 8254 so it generates a continuous series of pulses that have a high time of 100 μs and low time of 1 μs. Indicate the CLK frequency required to accomplish this task.
53. Why does a 50 percent duty cycle cause the motor to stand still in the motor speed and direction control circuit presented in this chapter?
54. What is asynchronous serial data?
55. What is synchronous serial data?
56. What is Baud rate?
57. Program the 8251A for asynchronous operation using 6 data bits, even parity, one stop bit, and a Baud rate divider of 1. Assume that the I/O ports are numbered 20H and 22H.
58. If the 8251A is to generate an asynchronous serial signal at a rate of 2400 Baud, and the divider is programmed for 16, what is the frequency of the signal attached to the TxC pin?
59. Describe the following terms: simplex, half-duplex, and full-duplex.
60. How is the 8251A reset?
61. The DAC0830 converts an 8-bit digital input to an analog output in approximately _____.
62. What is the step voltage at the output of the DAC0830 if the reference voltage is –2.55 V?
63. Interface a DAC0830 to the 8086 so it operates at I/O port 400H.

64. Develop a program using the interface of Question 62 so the DAC0830 generates a triangular voltage waveform. The frequency of this waveform must be approximately 100 Hz.
65. The ADC080X requires approximately _____ to convert an analog voltage into a digital code.
66. What is the purpose of the INTR pin on the ADC080X?
67. The $\overline{WR}$ pin on the ADC080X is used for what purpose?
68. Interface an ADC080X at I/O port 0260H for data and 0270H to test the INTR pin.
69. Develop a program for the ADC080X in Question 67 so it reads an input voltage once per 100 ms and stores the results in a memory array 100H bytes in length.

# CHAPTER 12

# Interrupts

## INTRODUCTION

In this chapter, we expand our coverage of basic I/O and programmable peripheral interfaces by examining a technique called interrupt-processed I/O. An interrupt is a hardware-initialed procedure that interrupts whatever program is currently executing.

This chapter provides examples and a detailed explanation of the interrupt structure of the entire Intel family of microprocessors.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Explain the interrupt structure of the Intel family of microprocessors.
2. Explain the operation of software interrupt instructions: INT, INTO, INT 3, and BOUND.
3. Explain how the interrupt enable flag bit (IF) modifies the interrupt structure.
4. Describe the function of the trap interrupt flag bit (TF) and the operation of trap-generated tracing.
5. Develop interrupt service procedures that control lower speed external peripheral devices.
6. Expand the interrupt structure of the microprocessor using the 8259A programmable interrupt controller and other techniques.
7. Explain the purpose and operation of a real-time clock.

## 12–1 BASIC INTERRUPT PROCESSING

In this section, we discuss the function of an interrupt in a microprocessor-based system and the structure and features of interrupts available to the Intel family of microprocessors.

## The Purpose of Interrupts

Interrupts are particularly useful when interfacing I/O devices that provide or require data at relatively low data transfer rates. In Chapter 11, for instance, we saw a keyboard example using strobed input operation of the 8255. In that example, software polled the 8255 and its IBF bit to decide whether data were available from the keyboard. If the person using the keyboard typed one character per second, the software for the 8255 waited an entire second between each keystroke for the person to type another key. This process is such a tremendous waste of time that designers have developed another process, called interrupt processing, to handle this situation.

Unlike the polling technique, interrupt processing allows the microprocessor to execute other software while the keyboard operator is thinking about what key to type next. As soon as a key is pressed, the keyboard encoder debounces the switch and puts out one pulse that interrupts the microprocessor. In this way, the microprocessor executes other software until a key is actually pressed; when it reads a key and returns to the program that was interrupted. As a result, the microprocessor can print reports, or complete any other task, while the operator is typing a document and thinking about what to type next.

Figure 12–1 shows a time line that indicates a typist typing data on a keyboard, a printer removing data from the memory, and a program executing. The program is the main program that is interrupted for each keystroke and each character that is to print on the printer. Note that the keyboard interrupt service procedure called by the keyboard interrupt and the printer interrupt service procedure each take little time to execute.

## Interrupts

The interrupts of the entire Intel family of microprocessors include two hardware pins that request interrupts (INTR and NMI) and one hardware pin ($\overline{\text{INTA}}$) that acknowledges the interrupt requested through INTR. In addition to the pins, the microprocessor also has software interrupts INT, INTO, INT 3, and BOUND. Two flag bits, IF (interrupt flag) and TF (trap flag), are also used with the interrupt structure and a special return instruction IRET (or IRETD in the 80386, 80486, or Pentium).

*Interrupt Vectors.* The interrupt vectors and vector table are crucial to an understanding of hardware and software interrupts. The *interrupt vector table* is located in the first 1,024 bytes of memory, at addresses 000000H–0003FFH. It contains 256 different four-byte interrupt vectors. An *interrupt vector* contains the address (segment and offset) of the interrupt service procedure.

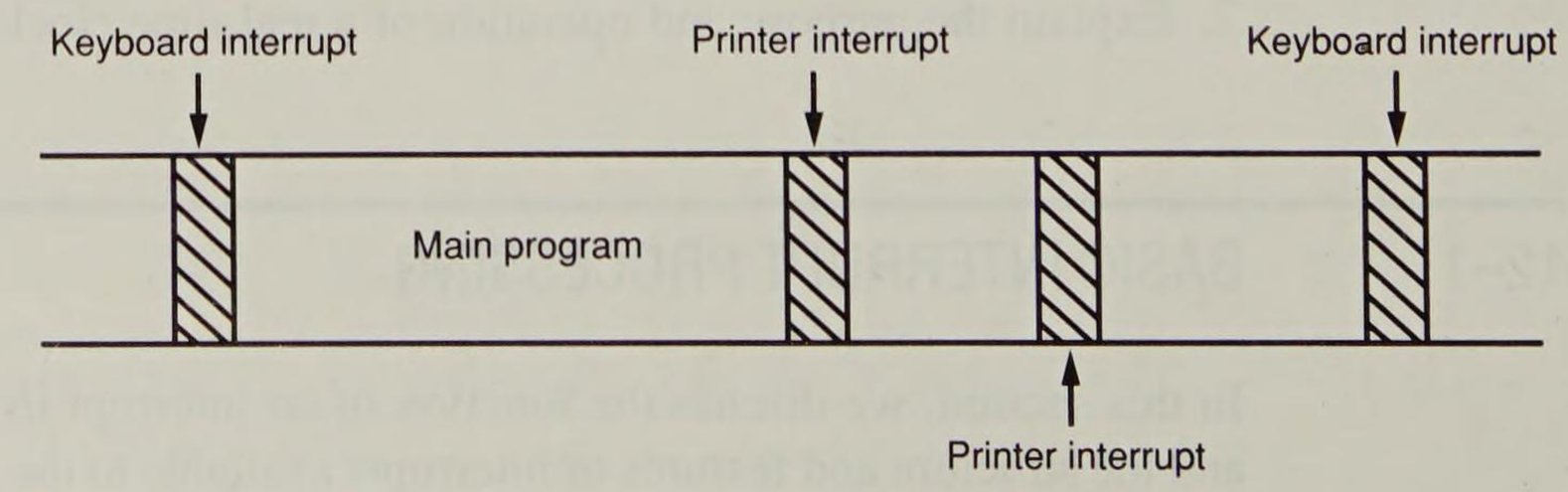



**FIGURE 12–1** A time line that indicates interrupt usage in a typical system.

Figure 12–2 illustrates the interrupt vector table for the microprocessor. The first 5 interrupt vectors are identical in all Intel microprocessor family members from the 8086 to the Pentium. Other interrupt vectors exist for the 80286 that are upward compatible to the 80386, 80486 and Pentium, but not downward compatible to the 8086 or 8088. Intel reserves the first 32 interrupt vectors for use in various microprocessor family members. The last 224 vectors are available as user interrupt vectors. Each vector is 4 bytes in length and contains the *starting address* of the interrupt service procedure. The first two bytes of the vector contain the offset address and the last two bytes contain the segment address.

The following list describes the function of each dedicated interrupt in the microprocessor:

Type 0 (Divide Error)—occurs whenever the result of a division overflows, or whenever an attempt is made to divide by zero.

Type 1 (Single-Step or Trap)—occurs after the execution of each instruction if the trap (TF) flag bit is set. Upon accepting this interrupt, the TF bit is cleared, so the interrupt service procedure executes at full speed. More detail is provided about this interrupt later in this section of the chapter.

Type 2 (Nonmaskable Hardware Interrupt)—a result of placing a logic 1 on the NMI input pin to the microprocessor. This input is nonmaskable, which means that it cannot be disabled.

Type 3 (One-Byte Interrupt)—a special one-byte instruction (INT 3) that uses this vector to access its interrupt service procedure. The INT 3 instruction is often used to store a breakpoint in a program for debugging.

Type 4 (Overflow)—a special vector used with the INTO instruction. The INTO instruction interrupts the program if an overflow condition exists, as reflected by the overflow flag (OF).

Type 5 (BOUND)—an instruction that compares a register with boundaries stored in the memory. If the content of the register is greater than or equal to the first word in memory and less than or equal to the second word, no interrupt occurs because the content of the register is within bounds. If the content of the register is out of bounds, a type 5 interrupt ensues.

Type 6 (Invalid Opcode)—occurs whenever an undefined opcode is encountered in a program.

Type 7 (Coprocessor Not Available)—occurs when a coprocessor is not found in the system, as dictated by the machine status word (MSW) coprocessor control bits. If an ESC or WAIT instruction executes and the coprocessor is not found, a type 7 exception or interrupt occurs.

Type 8 (Double Fault)—activated whenever two separate interrupts occur during the same instruction.

Type 9 (Coprocessor Segment Overrun)—occurs if the ESC instruction (coprocessor opcode) memory operand extends beyond offset address FFFFH.

Type 10 (Invalid Task State Segment)—occurs if the TSS is invalid because the segment limit field is not 002BH or higher. In most cases, this is because the TSS is not initialized.

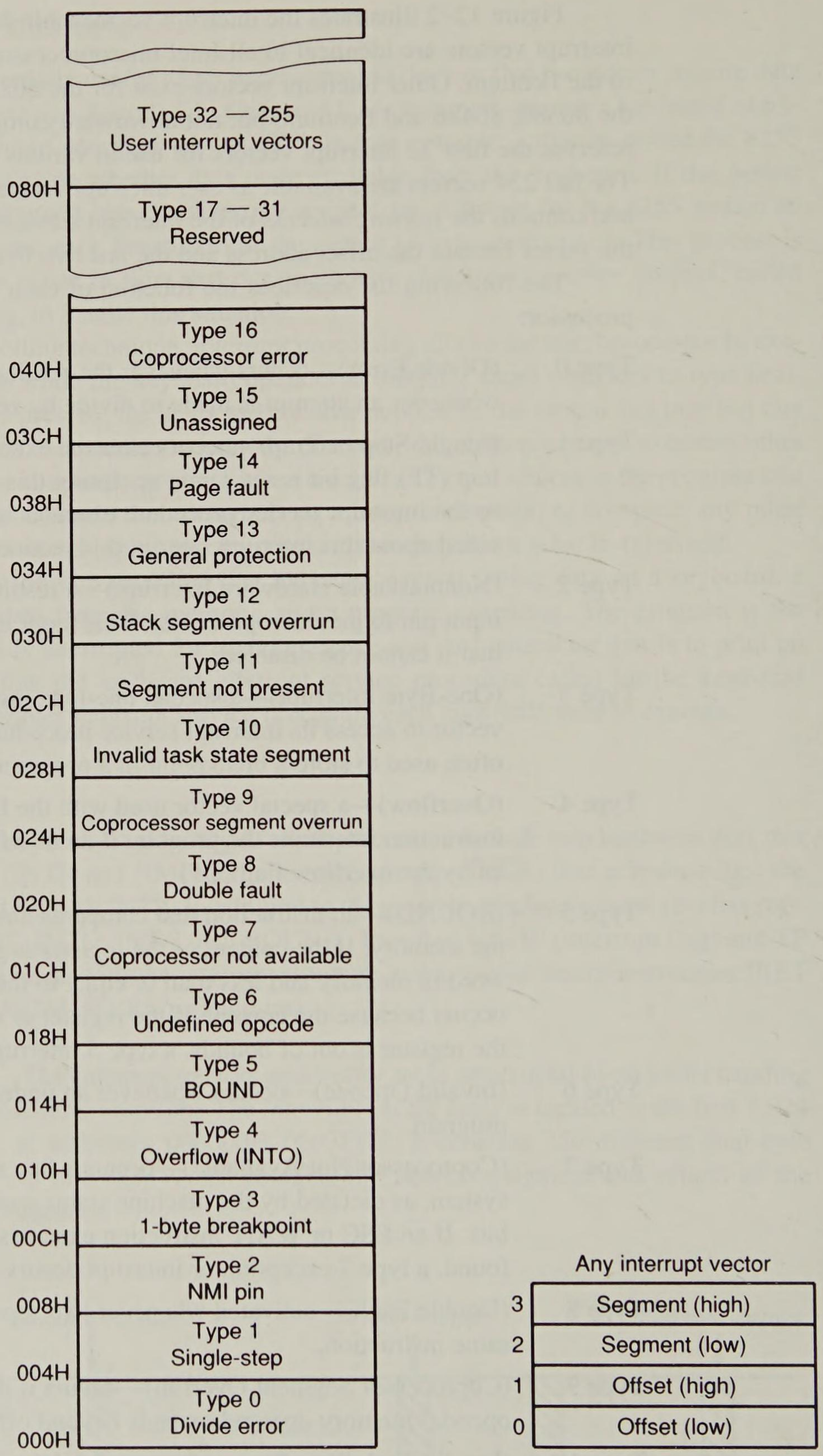



**FIGURE 12–2** (a) The interrupt vector table for the microprocessor. (b) The contents of an interrupt vector.

Type 11 (Segment Not Present)—occurs when the P bit in a descriptor indicates that the segment is not present or not valid (P = 0).

Type 12 (Stack Segment Overrun)—occurs if the stack segment is not present (P = 0) or if the limit of the stack segment is exceeded.

Type 13 (General Protection)—occurs for most protection violations in the 80286/80386/80486 protected-mode system. A list of these protection violations follows:

a. Descriptor table limit exceeded
b. Privilege rules violated
c. Invalid descriptor segment type loaded
d. Write to code segment that is protected
e. Read from execute-only code segment
f. Write to read-only data segment
g. Segment limit exceeded
h. CPL 0 ≠ when executing CTS, HLT, LGDT, LIDT, LLDT, LMSW, or LTR
i. CPL > IOPL when executing CLI, IN, INS, LOCK, OUT, OUTS, and STI

Type 14 (Page Fault)—occurs for any page fault memory or code access in the 80386, 80486, and Pentium microprocessors.

Type 16 (Coprocessor Error)—takes effect whenever a coprocessor error ($\overline{\text{ERROR}}$ = 0) occurs for the ESCape or WAIT instructions for the 80386, 80486, and Pentium microprocessors only.

Type 17 (Alignment Check)—indicates that word or doubleword data are addressed at an odd memory location, or incorrect location in the case of a doubleword. This interrupt is active in the 80486 and Pentium microprocessors.

Type 18 (Machine Check)—activates a system memory management mode interrupt in the Pentium microprocessor.

## Interrupt Instructions: BOUND, INTO, INT, INT 3, and IRET

Of the five software interrupt instructions available to the microprocessor, INT and INT 3 are very similar, BOUND and INTO are conditional, and IRET is a special interrupt return instruction.

The BOUND instruction, which has two operands, compares a register with two words of memory data. For example, if the instruction BOUND AX,DATA is executed, AX is compared with the contents of DATA and DATA+1, and also with DATA+2 and DATA+3. If AX is less than the contents of DATA and DATA+1, a type 5 interrupt occurs. If AX is greater than DATA+2 and DATA+3, a type 5 interrupt occurs. If AX is within the bounds of these two memory words, no interrupt occurs.

The INTO instruction checks the overflow flag (OF). If OF = 1, the INTO instruction calls the procedure, whose address is stored in interrupt vector type number 4. If OF = 0, then the INTO instruction performs a NOP and the next sequential instruction in the program executes.

The INT *n* instruction calls the interrupt service procedure that begins at the address represented in vector number *n*. For example, INT 80H or INT 128 call the interrupt service procedure whose address is stored in vector type number 80H (000200H–000203H). To determine the vector address, just multiply the vector type number (*n*) by 4. This gives the beginning address of the four-byte long interrupt vector. For example, an INT $5 = 4 \times 5 = 20$ or (14H). The vector for INT 5 begins at address 000014H and continues to 000017H. Each INT instruction is stored in two bytes of memory with the first byte containing the opcode and the second containing the interrupt type number. The only exception to this is the INT 3 instruction, a one-byte instruction. The INT 3 instruction is often used as a breakpoint interrupt, because it is easy to insert a 1-byte instruction into a program. Breakpoints are often used to debug faulty software.

The IRET instruction is a special return instruction used to return for both software and hardware interrupts. The IRET instruction is much like a normal far RET, because it retrieves the return address from the stack. It is unlike the normal return, because it also retrieves a copy of the flag register from the stack. An IRET instruction removes 6 bytes from the stack: 2 for the IP, 2 for the CS, and 2 for the flags.

In the 80386 and 80486 microprocessors, there is also an IRETD instruction, because these microprocessors can push the EFLAG register (32 bits) on the stack as well as the 32-bit EIP in the protected mode. If the 80386 and the 80486 are operated in the real mode, we use the IRET instruction.

## The Operation of a Real-Mode Interrupt

When the microprocessor completes executing the current instruction, it determines whether an interrupt is active by checking (1) instruction executions, (2) single-step, (3) NMI, (4) coprocessor segment overrun, (5) INTR, and (6) INT instruction in the order presented. If one or more of these interrupt conditions are present, the following sequence of events occurs:

1. The contents of the flag register are pushed onto the stack.
2. Both the interrupt (IF) and trap (TF) flags are cleared. This disables the INTR pin and the trap or single-step feature.
3. The contents of the code segment register (CS) are pushed onto the stack.
4. The contents of the instruction pointer (IP) are pushed onto the stack.
5. The interrupt vector contents are fetched and placed into both IP and CS, so the next instruction executes at the interrupt service procedure addressed by the vector.

Whenever an interrupt is accepted, the microprocessor stacks the contents of the flag register, CS and IP; clears both IF and TF; and jumps to the procedure addressed by the interrupt vector. After the flags are pushed onto the stack, IF and TF are cleared. These flags are returned to the state prior to the interrupt, when the IRET instruction is encountered at the end of the interrupt service procedure. Therefore, if interrupts were enabled prior to the interrupt service procedure, they are automatically reenabled by the IRET instruction at the end of the procedure.

The return address (in CS and IP) is pushed onto the stack during the interrupt. Sometimes the return address points to the next instruction in the program, and sometimes it points to the instruction or point in the program where the interrupt occurred. Interrupt

type numbers 0, 5, 6, 7, 8, 10, 11, 12, and 13 push a return address that points to the offending instruction, instead of the next instruction in the program. This allows the interrupt service procedure to possibly retry the instruction in certain error cases.

Some of the protected-mode interrupts (types 8, 10, 11, 12, and 13) place an error code on the stack following the return address. The error code identifies the selector that caused the interrupt. In cases where no selector is involved, the error code is a 0.

## Operation of a Protected-Mode Interrupt

In the protected mode, interrupts have exactly the same assignments as in real mode, but the interrupt vector table is different. In place of interrupt vectors, protected mode uses a set of 256 interrupt descriptors stored in an interrupt descriptor table (IDT). The interrupt descriptor table is 256 × 8 (2K) bytes in length, with each descriptor containing 8 bytes. The interrupt descriptor table is located at any memory location in the system by the interrupt descriptor table address register (IDTR).

Each entry in the IDT contains the address of the interrupt service procedure in the form of a segment selector and a 32-bit offset address. It also contains the P bit (present) and DPL bits to describe the privilege level of the interrupt. Figure 12–3 shows the contents of the interrupt descriptor.

Real-mode interrupt vectors can be converted into protected-mode interrupts by copying the interrupt procedure addresses from the interrupt vector table and converting them to 32-bit offset addresses that are stored in the interrupt descriptors. A single selector and segment descriptor can be placed in the global descriptor table that identifies the first 1M byte of memory as the interrupt segment.

Other than the IDT and interrupt descriptors, the protected-mode interrupt functions just like the real-mode interrupt. We return from both interrupts by using the IRET or IRETD instruction. The only difference is that in protected mode, the microprocessor accesses the IDT instead of the interrupt vector table.

## Interrupt Flag Bits

The interrupt flag (IF) and the trap flag (TF) are both cleared after the contents of the flag register are stacked during an interrupt. Figure 12–4 illustrates the contents of the flag register and the location of IF and TF. When the IF bit is set, it allows the INTR pin to cause

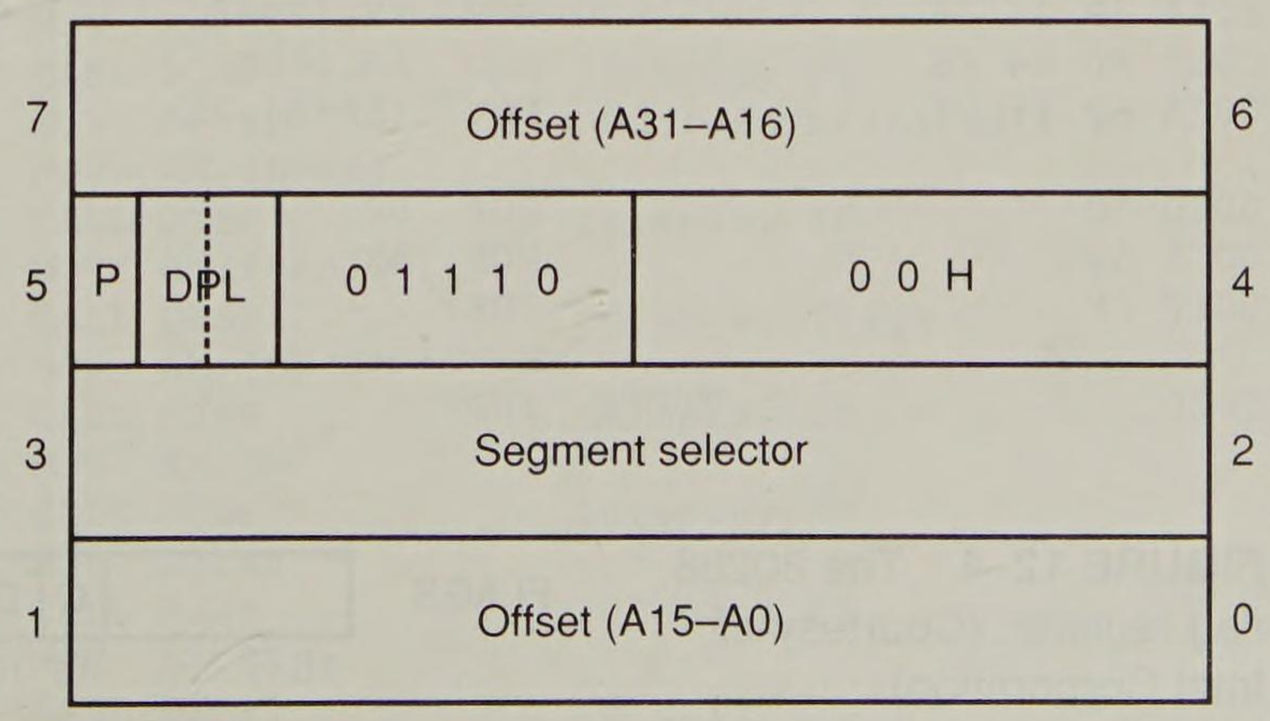



**FIGURE 12–3** The protected-mode interrupt descriptor.

an interrupt; when the IF bit is cleared, it prevents the INTR pin from causing an interrupt. When TF = 1, it causes a trap interrupt (type number 1) to occur after each instruction executes. This is why we often call a trap interrupt a single-step interrupt. When TF = 0, normal program execution occurs. This flag bit allows debugging in the 80386/80486 as explained in Chapter 9, or in the Pentium as will be explained in Chapter 14.

The interrupt flag is set and cleared by the STI and CLI instructions, respectively. There are no special instructions that set or clear the trap flag. Example 12–1 shows an interrupt service procedure that turns tracing on by setting the trap flag bit on the stack from inside the procedure. Example 12–2 shows an interrupt service procedure that turns tracing off by clearing the trap flag on the stack from within the procedure.

**EXAMPLE 12–1**

```
                        ;Procedure that sets TF to enable trap
                        ;
0000                    TRON    PROC FAR

0000 50                         PUSH AX                 ;save registers
0001 55                         PUSH BP

0002 8B EC                      MOV  BP,SP              ;get SP
0004 8B 46 08                   MOV  AX,[BP+8]          ;get flags
0007 80 CC 01                   OR   AH,1               ;set TF
000A 89 46 08                   MOV  [BP+8],AX          ;save flags

000D 5D                         POP  BP                 ;restore registers
000E 58                         POP  AX
000F CF                         IRET

0010                    TRON    ENDP
```

**EXAMPLE 12–2**

```
                        ;Procedure that clears TF to disable trap
                        ;
0010                    TROFF   PROC FAR

0010 50                         PUSH AX                 ;save registers
0011 55                         PUSH BP

0012 8B EC                      MOV  BP,SP              ;get SP
0014 8B 46 08                   MOV  AX,[BP+8]          ;get flags
0017 80 E4 FE                   AND  AH,0FEH            ;clear TF
001A 89 46 08                   MOV  [BP+8],AX          ;save flags

001D 5D                         POP  BP                 ;restore registers
001E 58                         POP  AX
001F CF                         IRET

0020                    TROFF   ENDP
```

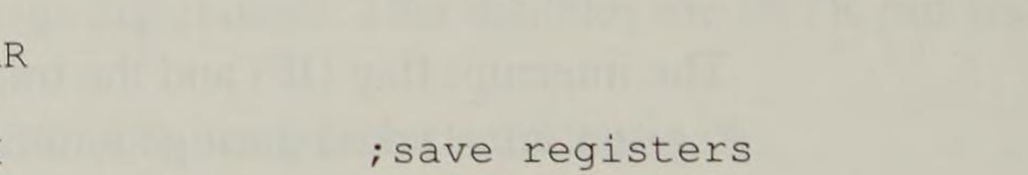



**FIGURE 12–4** The 80286 flag register. (Courtesy of Intel Corporation)

In both examples, the flag register is retrieved from the stack by using the BP register, which by default addresses the stack segment. After the flags are retrieved, the TF bit is either set (TRON) or cleared (TROFF) before returning from the interrupt service procedure. The IRET instruction restores the flag register with the new state of the trap flag.

*Trace Procedure.* Assuming that TRON is accessed by an INT 40H instruction and TROFF is accessed by an INT 41H instruction, Example 12–3 traces through a program immediately following the INT 40H instruction. The interrupt service procedure, illustrated in Example 12–3, responds to interrupt type number 1 or a trap interrupt. Each time this occurs—after each instruction following INT 40H executes—the TRACE procedure displays the contents of all the 16-bit microprocessor registers on the CRT screen. This provides a register trace of all the instructions between the INT 40H (TRON) and INT 41H (TROFF).

**EXAMPLE 12–3** (pp. 527–529)

```
                        ;Procedure that displays all the registers and their contents
                        ;on the video screen in response to a trap interrupt.
                        ;
0000                    TRACE  PROC FAR

0000 50                        PUSH AX                  ;save registers
0001 55                        PUSH BP
0002 53                        PUSH BX

0003 BB 0054 R                 MOV  BX,OFFSET NAMES ;address register names

                       ;display registers

0006 E8 009A R                 CALL CRLF                ;get new display line
0009 E8 00A9 R                 CALL DISP                ;display AX
000C 8B C3                     MOV  AX,BX
000E E8 00A9 R                 CALL DISP                ;display BX
0011 8B C1                     MOV  AX,CX
0013 E8 00A9 R                 CALL DISP                ;display CX
0016 8B C2                     MOV  AX,DX
0018 E8 00A9 R                 CALL DISP                ;display DX
001B 8B C4                     MOV  AX,SP
001D 05 000C                   ADD  AX,12
0020 E8 00A9 R                 CALL DISP                ;display SP
0023 8B C5                     MOV  AX,BP
0025 E8 00A9 R                 CALL DISP                ;display BP
0028 8B C6                     MOV  AX,SI
002A E8 00A9 R                 CALL DISP                ;display SI
002D 8B EC                     MOV  BP,SP               ;address stack with BP
002F 8B 46 06                  MOV  AX,[BP+6]
0032 E8 00A9 R                 CALL DISP                ;display IP
0035 8B 46 0A                  MOV  AX,[BP+10]
0038 E8 00A9 R                 CALL DISP                ;display flags
003B 8B 46 08                  MOV  AX,[BP+8]
003E E8 00A9 R                 CALL DISP                ;display CS
0041 8C D8                     MOV  AX,DS
0043 E8 00A9 R                 CALL DISP                ;display DS
0046 8C C0                     MOV  AX,ES
0048 E8 00A9 R                 CALL DISP                ;display ES
004B 8C D0                     MOV  AX,SS
004D E8 00A9 R                 CALL DISP                ;display SS
```

```
0050 5B                         POP  BX                ;restore register
0051 5D                         POP  BP
0052 58                         POP  AX
0053 CF                         IRET

0054                    TRACE   ENDP

0054 41 58 20 3D 20     NAMES   DB   'AX = '
0059 42 58 20 3D 20             DB   'BX = '
005E 43 58 20 3D 20             DB   'CX = '
0063 44 58 20 3D 20             DB   'DX = '
0068 53 50 20 3D 20             DB   'SP = '
006D 42 50 20 3D 20             DB   'BP = '
0072 53 49 20 3D 20             DB   'SI = '
0077 44 49 20 3D 20             DB   'DI = '
007C 49 50 20 3D 20             DB   'IP = '
0081 46 4C 20 3D 20             DB   'FL = '
0086 43 53 20 3D 20             DB   'CS = '
008B 44 53 20 3D 20             DB   'DS = '
0090 45 53 20 3D 20             DB   'ES = '
0095 53 53 20 3D 20             DB   'SS = '

009A                    CRLF    PROC NEAR

009A 50                         PUSH AX                ;save registers
009B 52                         PUSH DX

009C B4 06                      MOV  AH,6
009E B2 0D                      MOV  DL,13
00A0 CD 21                      INT  21H               ;display CR
00A2 B2 0A                      MOV  DL,10
00A4 CD 21                      INT  21H               ;display LF

00A6 5A                         POP  DX                ;restore registers
00A7 58                         POP  AX
00A8 C3                         RET

00A9                    CRLF    ENDP

00A9                    DISP    PROC NEAR

00A9 52                         PUSH DX                ;save registers
00AA 57                         PUSH DI
00AB 51                         PUSH CX
00AC 50                         PUSH AX

00AD B4 06                      MOV  AH,6
00AF B9 0005                    MOV  CX,5

00B2                    DISP1:

00B2 2E: 8A 17                  MOV  DL,CS:[BX]        ;display name
00B5 CD 21                      INT  21H
00B7 43                         INC  BX
00B8 E2 F8                      LOOP DISP1
00BA 5F                         POP  DI                ;get numeric value
00BB 57                         PUSH DI
00BC B6 04                      MOV  DH,4              ;set count

00BE                    DISP2:

00BE B9 0004                    MOV  CX,4
```

```
00C1 D3 C7                  ROL   DI,CL
00C3 8B C7                  MOV   AX,DI
00C5 B4 06                  MOV   AH,6
00C7 8A D0                  MOV   DL,AL
00C9 80 E2 0F               AND   DL,15
00CC 80 C2 30               ADD   DL,30H          ;convert to ASCII
00CF 80 FA 39               CMP   DL,39H
00D2 76 03                  JBE   DISP3
00D4 80 C2 07               ADD   DL,7

00D7                  DISP3:

00D7 CD 21                  INT   21H
00D9 FE CE                  DEC   DH
00DB 75 E1                  JNZ   DISP2           ;repeat for 4 digits

00DD B4 06                  MOV   AH,6
00DF B2 20                  MOV   DL,' '          ;display space
00E1 CD 21                  INT   21H

00E3 58                     POP   AX              ;restore registers
00E4 59                     POP   CX
00E5 5F                     POP   DI
00E6 5A                     POP   DX
00E7 C3                     RET

00E8                  DISP  ENDP
```

## Storing an Interrupt Vector in the Vector Table

In order to install an interrupt vector—sometimes called a *hook*—the assembler must address absolute memory. Example 12–4 shows how a new vector is added to the interrupt vector table by using the assembler and a DOS function call. Here, INT 21H function call number 25H initializes the interrupt vector. The first thing done in this procedure is to save the old interrupt vector number using DOS INT 21H function call number 35H to read the current vector. Refer to Appendix A for more detail on DOS function calls.

**EXAMPLE 12–4**

```
                        ;Procedure that installs a new interrupt vector at interrupt
                        ;type number 40H
                        ;
0000 0000               OLDOFF DW   ?
0002 0000               OLDSEG DW   ?
= 0004                  NEWOFF EQU  THIS WORD
= 0006                  NEWSEG EQU  THIS WORD+2
0004 0200 ---- R        NEW    DD   FAR PTR TRON

0008                    IN_40H PROC NEAR

0008 06                        PUSH ES              ;save registers
0009 1E                        PUSH DS
000A 50                        PUSH AX
000B 53                        PUSH BX
000C 52                        PUSH DX

000D B4 35                     MOV  AH,35H          ;get current vector
000F B0 40                     MOV  AL,40H          ;type number
0011 CD 21                     INT  21H
```

```
0013 2E: 89 1E 0000 R          MOV  CS:OLDOFF,BX     ;save old offset
0018 8C C0                     MOV  AX,ES
001A 2E: A3 0002 R             MOV  CS:OLDSEG,AX     ;save old segment

001E 2E: 8B 16 0004 R          MOV  DX,NEWOFF        ;get new offset
0023 2E: A1 0006 R             MOV  AX,NEWSEG        ;get new segment
0027 8E D8                     MOV  DS,AX

0029 B4 25                     MOV  AH,25H           ;install new vector
002B B0 40                     MOV  AL,40H           ;type number
002D CD 21                     INT  21H

002F 5A                        POP  DX               ;restore registers
0030 5B                        POP  BX
0031 58                        POP  AX
0032 1F                        POP  DS
0033 07                        POP  ES
0034 C3                        RET

0035                    IN_40H ENDP
```

## 12–2 HARDWARE INTERRUPTS

The microprocessor has two hardware interrupt inputs: nonmaskable interrupt (NMI) and interrupt request (INTR). Whenever the NMI input is activated, a type 2 interrupt occurs, because NMI is internally decoded. The INTR input must be externally decoded to select a vector. Any interrupt vector can be chosen for the INTR pin, but we usually use an interrupt type number between 20H and FFH. The $\overline{\text{INTA}}$ signal is also an interrupt pin on the microprocessor, but it is an output that is used, in response to the INTR input, to apply a vector type number to the data bus connections D7–D0. Figure 12–5 shows the three user interrupt connections on the microprocessor.

### NMI

The nonmaskable interrupt (NMI) is an edge-triggered input that requests an interrupt on the positive edge (0-to-1) transition. After a positive edge, the NMI pin must remain a

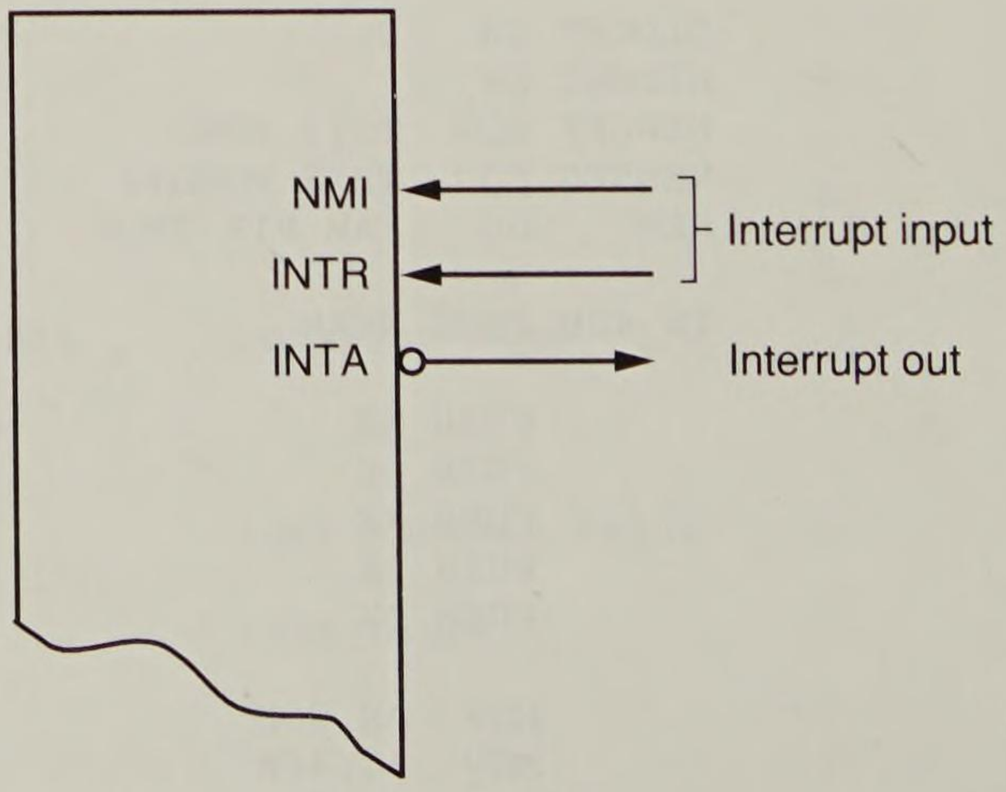



**FIGURE 12–5** The 80X86 interrupt pins.

logic 1 until it is recognized by the microprocessor. Note that before the positive edge is recognized, the NMI pin must be a logic 0 for at least 2 clocking periods.

The NMI input is often used for parity errors and other major system faults, such as power failures. Power failures are easily detected by monitoring the AC power line and causing an NMI interrupt whenever AC power drops out. In response to this type of interrupt, the microprocessor stores all the internal register in a battery-backed-up memory or an EEPROM. Figure 12–6 shows a power failure detection circuit that provides a logic 1 to the NMI input whenever AC power is interrupted.

In this circuit, an optical isolator provides isolation from the AC power line. The output of the isolator is shaped by a Schmitt-trigger inverter that provides a 60 Hz pulse to the trigger input of the 74LS122 retriggerable monostable multivibrator. The values of R and C are chosen, so the 74LS122 has an active pulse width of 33 ms or 2 AC input periods. Because the 74LS122 is retriggerable, the Q output remains triggered at a logic 1 and $\overline{Q}$ remains a logic 0, as long as AC power is applied.

If the AC power fails, the 74LS122 no longer receives trigger pulses from the 74ALS14, which means Q returns to a logic 0 and $\overline{Q}$ returns to a logic 1, interrupting the microprocessor through the NMI pin. The interrupt service procedure, not shown here, stores the contents of all internal registers and other data into a battery-backed-up memory. This system assumes that the system power supply has a large enough filter capacitor to provide energy for at least 75 ms after the AC power ceases.

Figure 12–7 shows a circuit that supplies power to a memory after the DC power fails. Here, diodes are used to switch supply voltages from the DC power supply to the battery. The diodes used are standard silicon diodes, because the power supply to this memory circuit is elevated above +5.0 V, to +5.7 V. The resistor is used to trickle-charge the battery which is either NiCAD, lithium, or a gel cell.

When DC power fails, the battery provides a reduced voltage to the Vcc connection on the memory device. Most memory devices will retain data with Vcc voltages as low as 1.5 V, so the battery voltage does not need to be +5.0 V. The $\overline{WR}$ pin is pulled to Vcc during a power outage, so no data will be written to the memory.

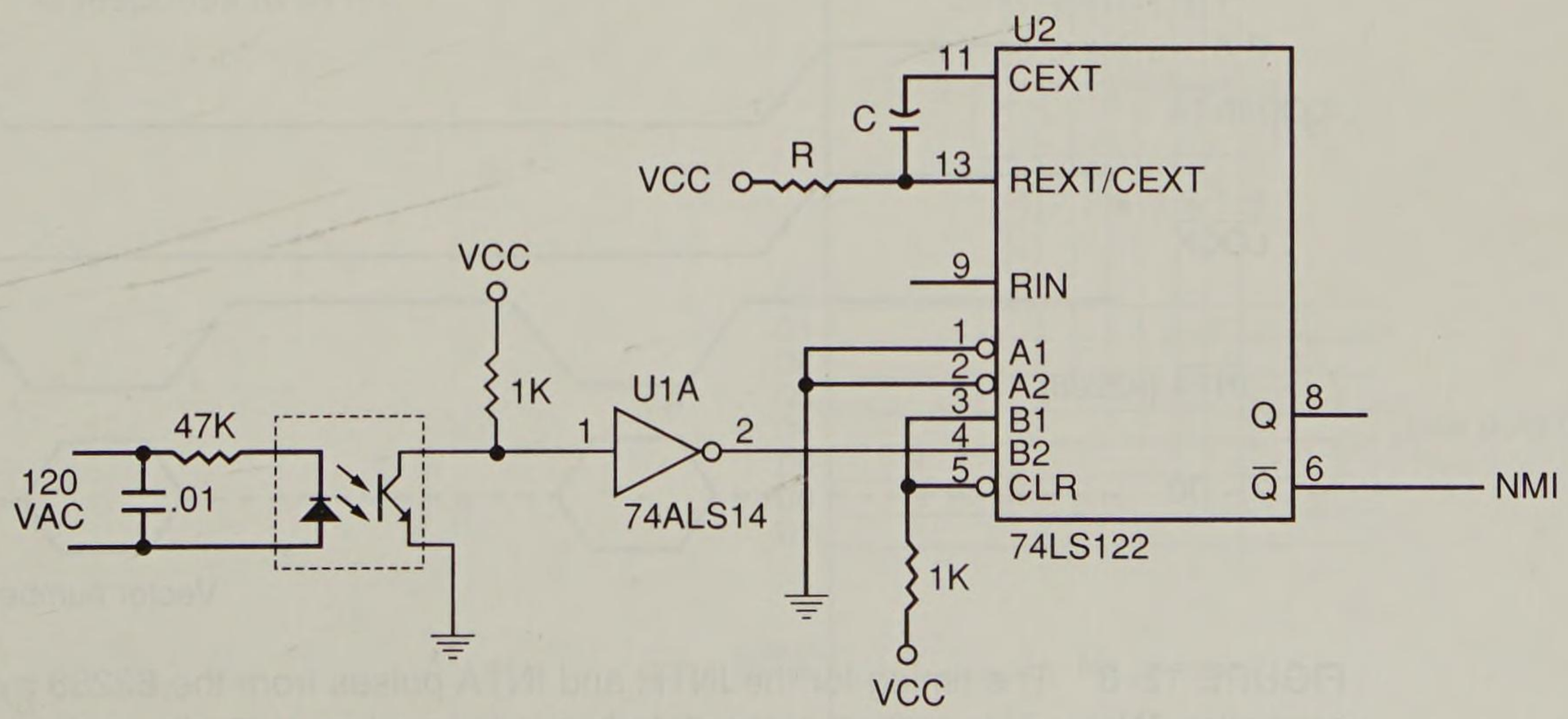



**FIGURE 12–6** A power failure detection circuit.

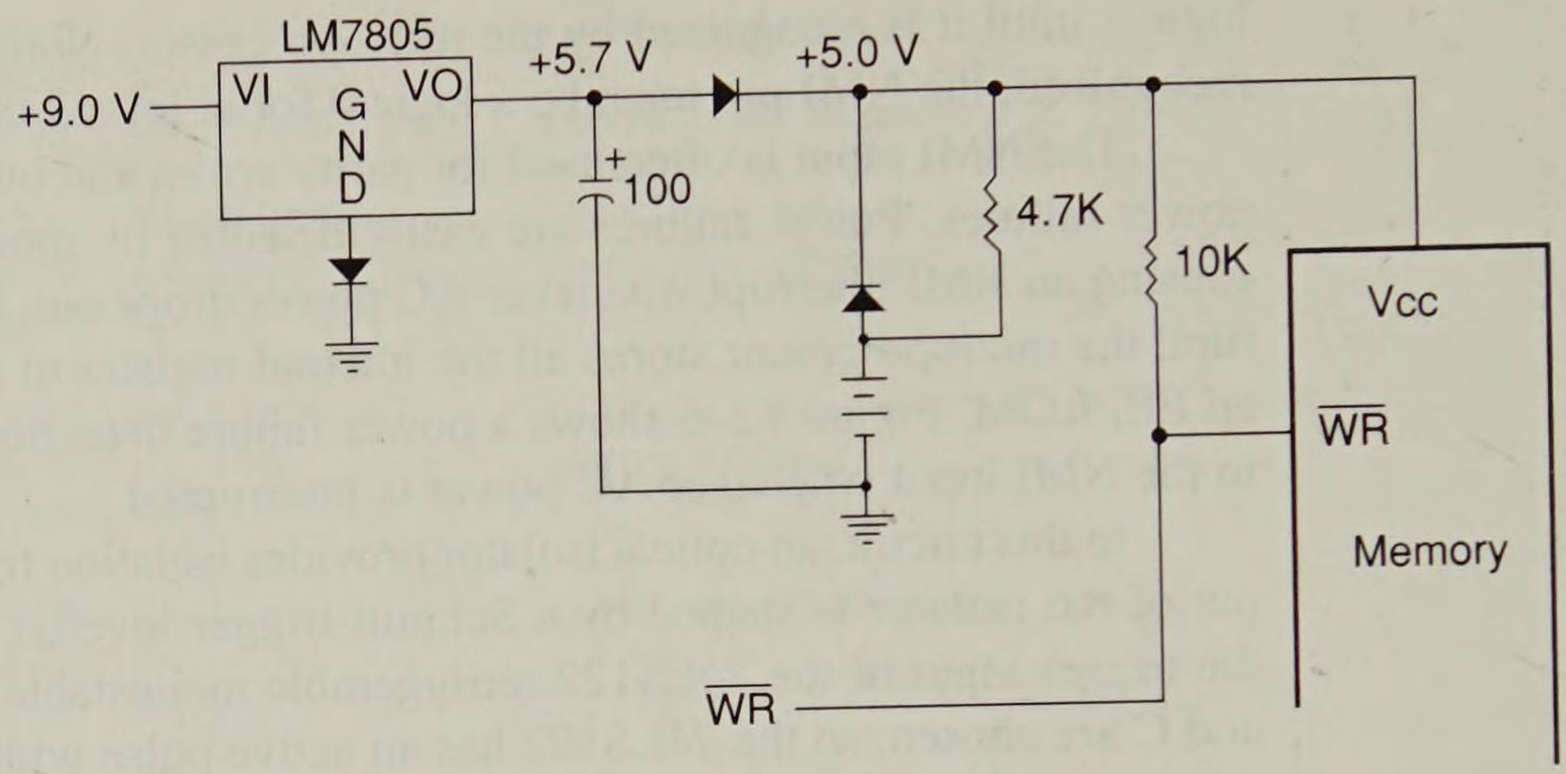



**FIGURE 12–7** A battery-backed-up memory system using a NiCad, lithium, or gel cell.

## INTR and $\overline{\text{INTA}}$

The interrupt request input (INTR) is level-sensitive, which means that it must be held at a logic 1 level until it is recognized. The INTR pin is set by an external event and cleared inside the interrupt service procedure. This input is automatically disabled once it is accepted by the microprocessor and reenabled by the IRET instruction at the end of the interrupt service procedure. The 80386, 80486, and Pentium use the IRETD instruction in the protected mode of operation.

The microprocessor responds to the INTR input by pulsing the $\overline{\text{INTA}}$ output in anticipation of receiving an interrupt vector type number on data bus connection D7–D0. Figure 12–8 shows the timing diagram for the INTR and $\overline{\text{INTA}}$ pins of the microprocessor. There are two $\overline{\text{INTA}}$ pulses, used to insert the vector type number on the data bus, that are generated by the system.

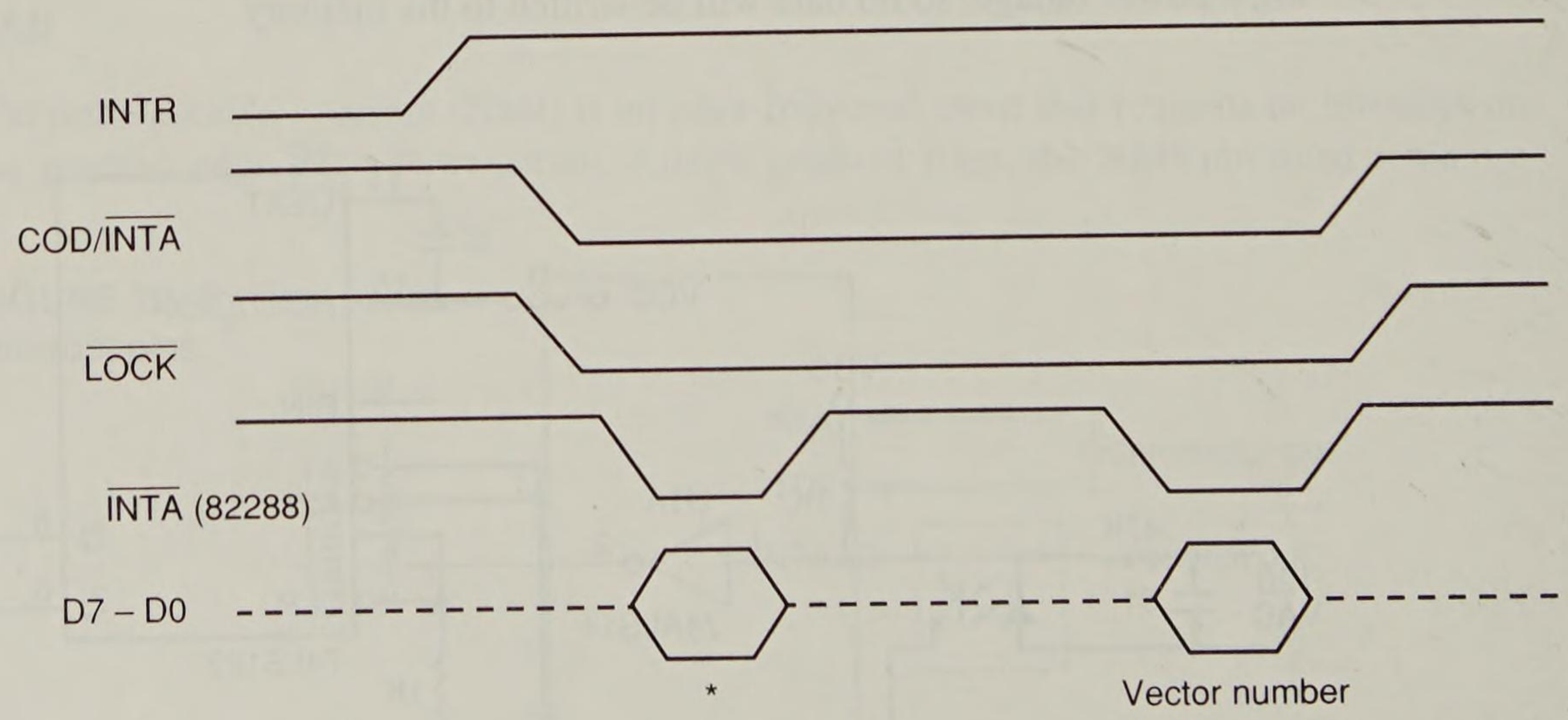



**FIGURE 12–8** The timing for the INTR and INTA pulses from the 82288 system bus controller. *Note: This portion of the data bus is ignored and usually contains the vector type number.

Figure 12–9 illustrates a simple circuit that applies interrupt vector type number FFH to the data bus in response to an INTR. Notice that the $\overline{\text{INTA}}$ pin is not connected in this circuit. Because resistors are used to pull the data bus connections (D0—D7) high, the microprocessor automatically sees vector type number FFH in response to the INTR input. This is possibly the least expensive way to implement the INTR pin on the microprocessor.

*Using a Three-State Buffer for INTA.* Figure 12–10 shows how interrupt vector type number 80H is applied to the data bus (D0–D7) in response to an INTR. In response to the INTR, the microprocessor outputs the $\overline{\text{INTA}}$, which is used to enable a 74ALS244 three-state octal buffer. The octal buffer applies the interrupt vector type number to the data bus in response to the $\overline{\text{INTA}}$ pulse. The vector type number is easily changed with the DIP switches that are shown in Figure 12–10.

*Making the INTR Input Edge-Triggered.* Often we need an edge-triggered input instead of a level-sensitive input. The INTR input can be converted to an edge-triggered input by using a D-type flip-flop, as illustrated in Figure 12–11. Here, the clock input becomes an edge-triggered interrupt request input, and the clear input is used to clear the request, when the $\overline{\text{INTA}}$ signal is output by the microprocessor. Note that the RESET signal initially clears the flip-flop, so no interrupt is requested when the system is first powered.

## The 8255 Keyboard Interrupt

The keyboard example presented in the last chapter provides a simple example of the operation of the INTR input and an interrupt. Figure 12–12 illustrates the interconnection of the 8255 with the microprocessor and the keyboard. It also shows how a 74ALS244 octal buffer is used to provide the microprocessor with interrupt vector type number 40H, in response to the keyboard interrupt during the $\overline{\text{INTA}}$ pulse.

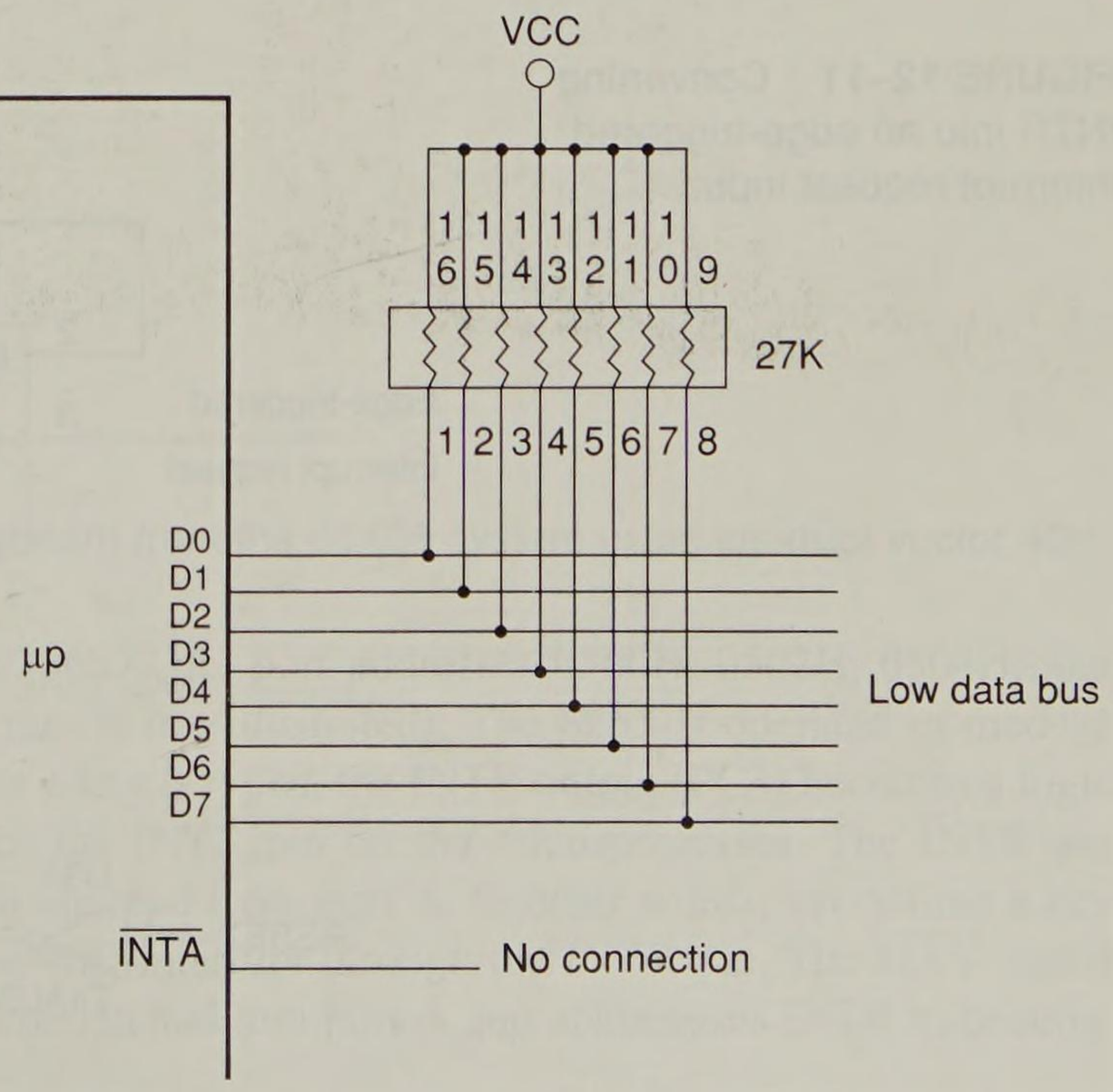



**FIGURE 12–9** A simple method for generating interrupt vector type number FFH in response to INTR.

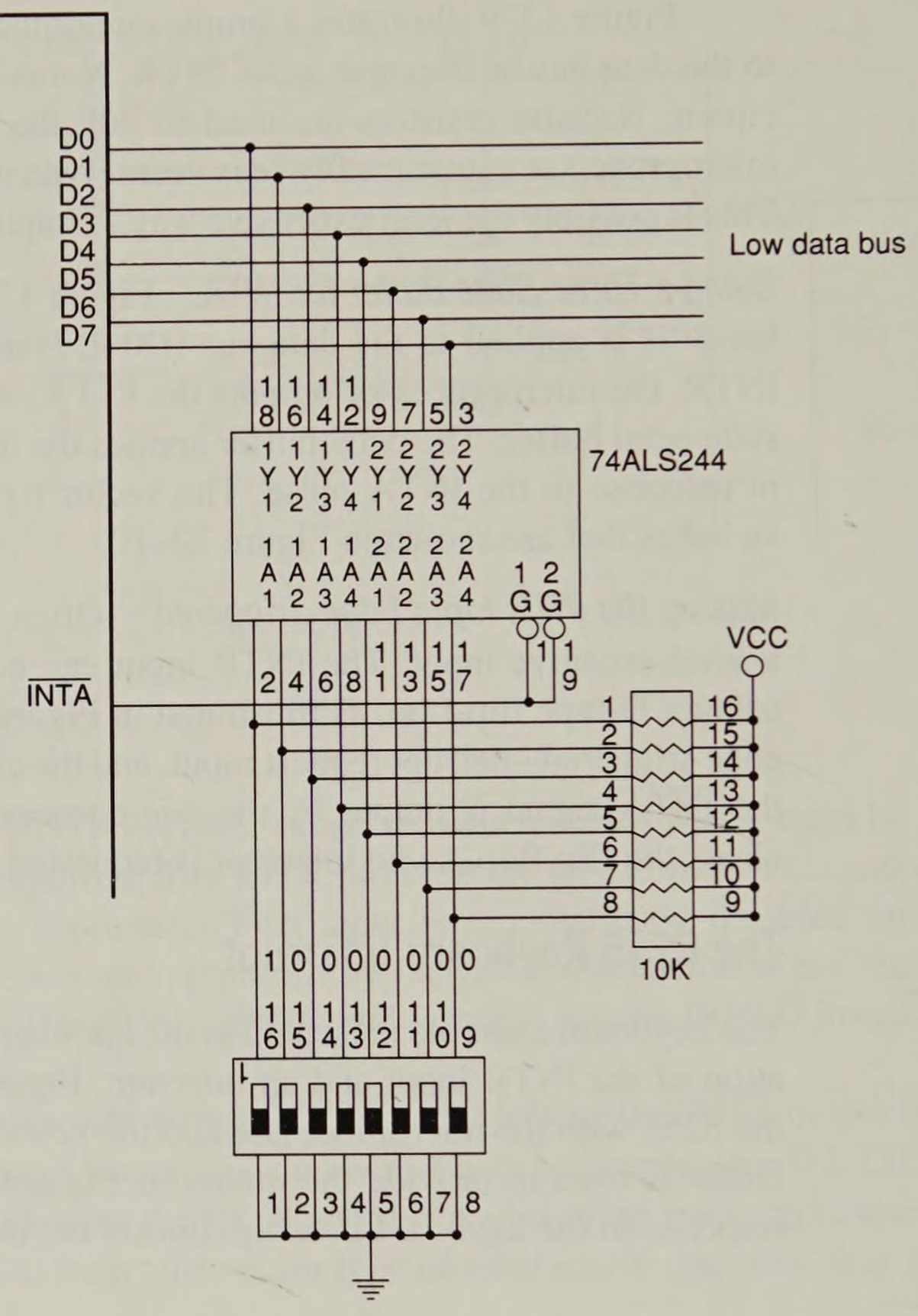



**FIGURE 12–10** A circuit that applies any interrupt vector type number in response to INTA. Here the circuit is applying type number 80H.

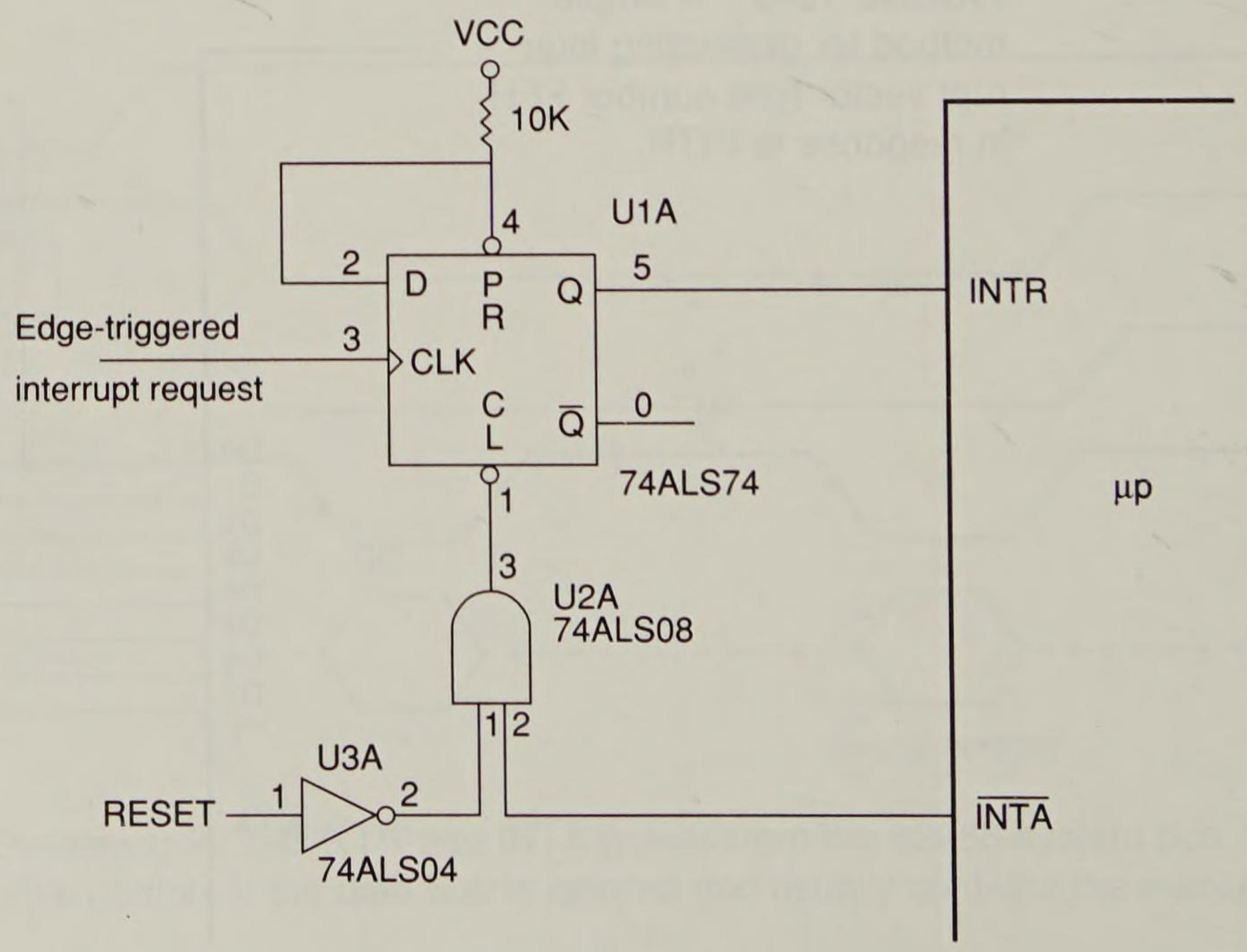



**FIGURE 12–11** Converting INTR into an edge-triggered interrupt request input.

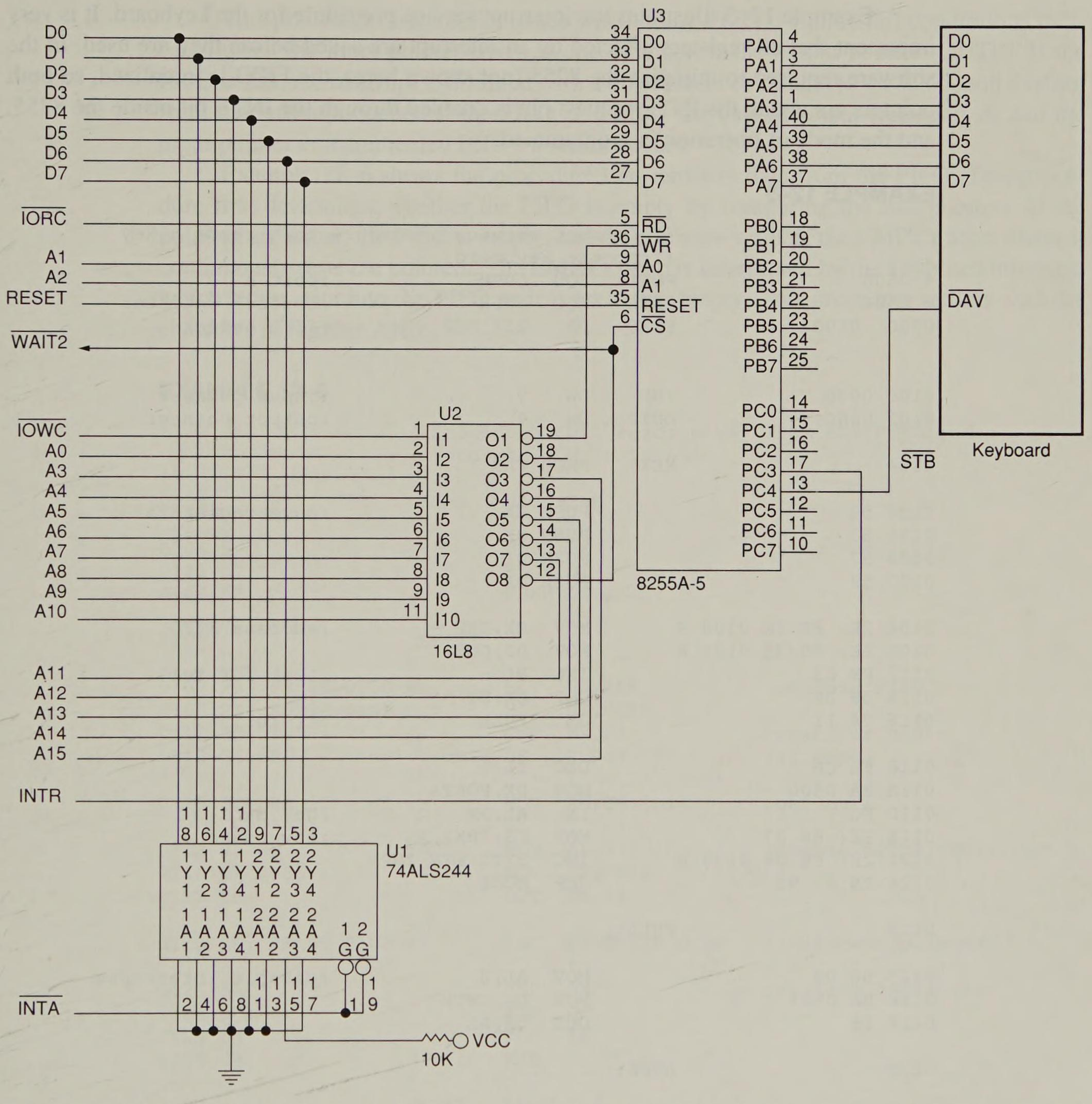



**FIGURE 12–12** An 8255A-5 interfaced to a keyboard from the 80X86 system using interrupt vector 40H.

The 8255 is decoded at 80386SX I/O port addresses 0500H, 0502H, 0504H, and 0506H by a PAL16L8 (the program is not illustrated). The 8255 is operated in mode 1 (strobed input mode), so whenever a key is typed, the INTR output (PC3) becomes a logic 1, requesting an interrupt through the INTR pin on the microprocessor. The INTR pin remains high until the ASCII data are read from port A. In other words, every time a key is typed, the 8255 requests a type 40H interrupt through the INTR pin. The $\overline{\text{DAV}}$ signal from the keyboard causes data to be latched into port A and also causes INTR to become a logic 1.

Example 12–5 illustrates the interrupt service procedure for the keyboard. It is very important that all registers affected by an interrupt are saved before they are used. In the software required to initialize the 8255 (not shown here), the FIFO is initialized, so both pointers are equal, the INTR request pin is enabled through the INTE bit inside the 8255, and the mode of operation is programmed.

**EXAMPLE 12–5**

```
                        ;Interrupt service procedure that reads a key
                        ;from the keyboard.
= 0500                  PORTA   EQU  500H            ;port A
= 0506                  CNTR    EQU  506H            ;control register
0000  0100[             FIFO    DB   256 DUP (?)     ;FIFO buffer
            ??
               ]
0100 0000               INP     DW   ?               ;input pointer
0102 0000               OUTP    DW   ?               ;output pointer

0104                    KEY     PROC FAR

0104 50                         PUSH AX              ;save registers
0105 53                         PUSH BX
0106 57                         PUSH DI
0107 52                         PUSH DX

0108 2E: 8B 1E 0100 R           MOV  BX,INP          ;address FIFO
010D 2E: 8B 3E 0102 R           MOV  DI,OUTP
0112 FE C3                      INC  BL              ;test for full
0114 3B DF                      CMP  BX,DI
0116 74 11                      JE   FULL            ;if full

0118 FE CB                      DEC  BL
011A BA 0500                    MOV  DX,PORTA
011D EC                         IN   AL,DX           ;get data
011E 2E: 88 07                  MOV  CS:[BX],AL      ;save data
0121 2E: FE 06 0100 R           INC  BYTE PTR INP
0126 EB 07 90                   JMP  DONE

0129                    FULL:

0129 B0 08                      MOV  AL,8            ;disable interrupts
012B BA 0506                    MOV  DX,CNTR
012E EE                         OUT  DX,AL

012F                    DONE:

012F 5A                         POP  DX              ;restore registers
0130 5F                         POP  DI
0131 5B                         POP  BX
0132 58                         POP  AX
0133 CF                         IRET

0134                    KEY     ENDP
```

The procedure is fairly short because the 80386SX already knows that keyboard data are available when the procedure is called. Data are input from the keyboard and then stored in the FIFO (first-in, first-out) buffer. Most keyboard interfaces contain a FIFO that is at least 16 bytes in depth. The FIFO in this example is 256 bytes deep, which is more than adequate for a keyboard interface.

This procedure first checks to see whether the FIFO is full. A full condition is indicated when the input pointer (INP) is one byte below the output pointer (OUTP). If the FIFO is full, the interrupt is disabled with a bit set/reset command to the 8255, and a return from the interrupt occurs. If the FIFO is not full, the data are input from port A, and the input pointer is incremented before a return occurs.

Example 12–6 shows the procedure that removes data from the FIFO. This procedure first determines whether the FIFO is empty by comparing the two pointers. If the pointers are equal, the FIFO is empty, and the software waits at the EMPTY loop where it continuously tests the pointers. The EMPTY loop is interrupted by the keyboard interrupt, which stores data into the FIFO so it is no longer empty. This procedure returns with the character in register AH.

**EXAMPLE 12–6**

```
                        ;Procedure that reads a key from the FIFO and
                        ;returns with it in AH
                        ;
0104                    READ    PROC FAR

0104 53                         PUSH BX                 ;save registers
0105 57                         PUSH DI
0106 52                         PUSH DX

0107                    EMPTY:

0107 2E: 8B 1E 0100 R           MOV  BX,INP             ;address FIFO
010C 2E: 8B 3E 0102 R           MOV  DI,OUTP
0111 3B DF                      CMP  BX,DI              ;test for empty
0113 74 F2                      JE   EMPTY              ;if empty

0115 2E: 8A 25                  MOV  AH,CS:[DI]         ;get data

0118 B0 09                      MOV  AL,9               ;enable 8255A interrupt
011A BA 0506                    MOV  DX,CNTR
011D EE                         OUT  DX,AL

011E 2E: FE 06 0102 R           INC  BYTE PTR OUTP      ;increment pointer

0123 5A                         POP  DX                 ;restore registers
0124 5F                         POP  DI
0125 5B                         POP  BX
0126 CB                         RET

0127                    READ    ENDP
```

## 12–3 EXPANDING THE INTERRUPT STRUCTURE

This text covers three common methods of expanding the interrupt structure of the microprocessor. In this section, we explain how, with software and some hardware modification of the circuit of Figure 12–10, it is possible to expand the INTR input so that it accepts 7 interrupt inputs. We also explain how to daisy-chain interrupts by software polling. In the next section, we describe a third technique, in which up to 63 interrupting inputs can be added by means of the 8259A programmable interrupt controller.

## Using the 74ALS244 to Expand Interrupts

The modification shown in Figure 12–13 allows the circuit in Figure 12–10 to accommodate up to seven additional interrupt inputs. The only hardware change is the addition of an 8-input NAND gate, which provides the INTR signal to the microprocessor when any of the $\overline{IR}$ inputs become active.

*Operation.* If any of the $\overline{IR}$ inputs become a logic 0, then the output of the NAND gate goes to a logic 1 and requests an interrupt through the INTR input. Which interrupt vector is fetched during the $\overline{INTA}$ pulse depends on which interrupt request line becomes active. Table 12–1 shows the interrupt vectors used by a single interrupt request input.

If two or more interrupt request inputs are simultaneously active, a new interrupt vector is generated. For example, if $\overline{IR1}$ and $\overline{IR0}$ are both active, the interrupt vector generated is FCH (252). Priority is resolved at this location. If the $\overline{IR0}$ input is to have the higher priority, the vector address for $\overline{IR0}$ is stored at vector location FCH. The entire top half of the vector table and its 128 interrupt vectors must be used to accommodate all possible conditions of these seven interrupt request inputs. This seems wasteful, but in many dedicated applications it is a cost-effective approach to interrupt expansion.

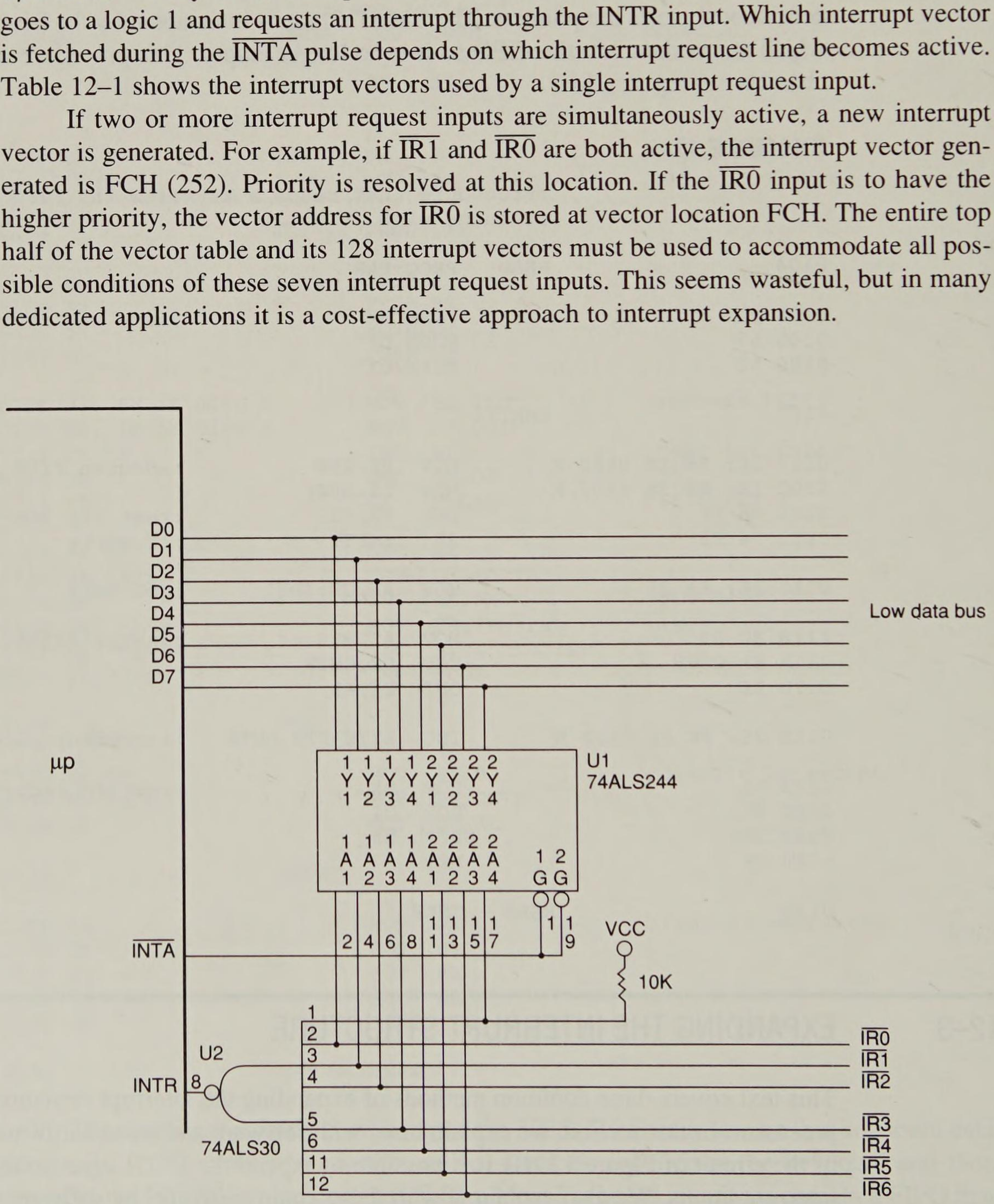



**FIGURE 12–13** Expanding the INTR input from one to seven interrupt request lines.

**TABLE 12–1** Single interrupt request for Figure 12–13

| $\overline{AD7}$ | $\overline{IR6}$ | $\overline{IR5}$ | $\overline{IR4}$ | $\overline{IR3}$ | $\overline{IR2}$ | $\overline{IR1}$ | $\overline{IR0}$ | *Vector* |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | FEH (254) |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | FDH (253) |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | FBH (251) |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | F7H (247) |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | EFH (239) |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | DFH (223) |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | BFH (191) |

## Daisy-Chained Interrupt

Expansion by means of a daisy-chained interrupt is in many ways better than using the 74ALS244 interrupt expansion, because it requires only one interrupt vector. The task of determining priority is left to the interrupt service procedure. Setting priority for a daisy chain requires additional software execution time but, in general, this is a much better approach to expanding the interrupt structure of the microprocessor.

Figure 12–14 illustrates a set of two 8255 peripheral interfaces with their four INTR outputs daisy-chained and connected to the single INTR input of the microprocessor. If any interrupt output becomes a logic 1, so does the INTR input to the microprocessor, causing an interrupt.

When a daisy chain is used to request an interrupt, it is better to pull the data bus connections (D0–D7) high using pullup resisters, so interrupt vector FFH is used for the chain. Actually, any interrupt vector can be used to respond to a daisy chain. In the circuit, any of the four INTR outputs from the two 8255s will cause the INTR pin on the microprocessor to go high, requesting an interrupt.

When the INTR pin does go high with a daisy chain, the hardware gives no direct indication which 8255 or INTR output caused the interrupt. This task is up to the interrupt service procedure, which must poll the 8255s to determine which INTR output became active, causing the interrupt.

Example 12–7 illustrates the interrupt service procedure that responds to the daisy-chain interrupt request. This procedure polls each 8255 and each INTR output to decide which interrupt service procedure to utilize.

**EXAMPLE 12–7**

```
                        ;Procedure that services the daisy-chain interrupt
                        ;
= 0504                  C1      EQU  504H           ;first 8255A
= 0604                  C2      EQU  604H           ;second 8255A
= 0001                  MASK1   EQU  1              ;INTRB
= 0008                  MASK2   EQU  8              ;INTRA

0000                    POLL    PROC FAR

0000 50                         PUSH AX             ;save registers
0001 52                         PUSH DX

0002 BA 0504                    MOV  DX,C1          ;address port C
```

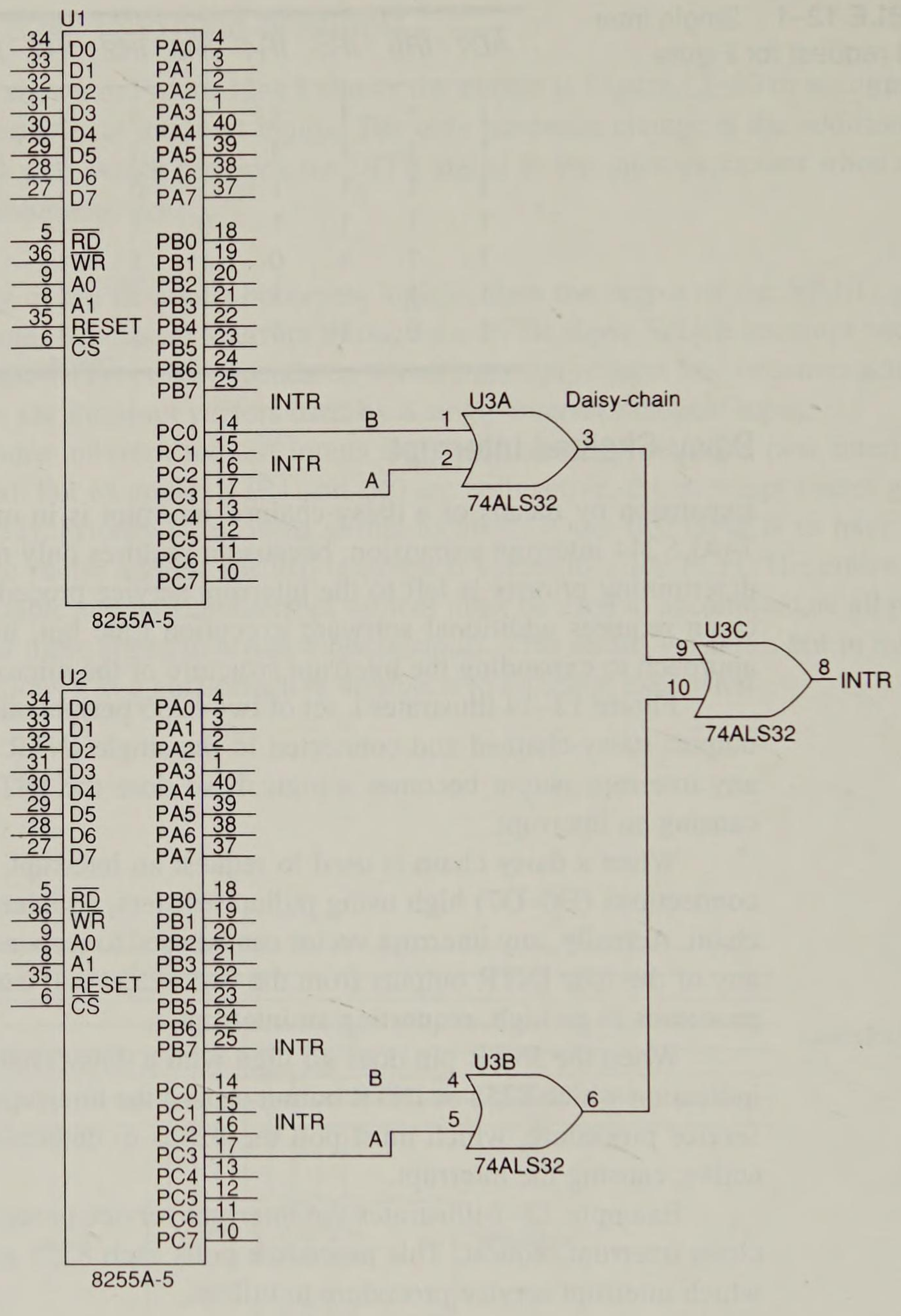



**FIGURE 12–14** Two 8255A-5 PIAs connected to the INTR outputs are daisy-chained to produce an INTR signal for the 80X86.

```
0005 EC                         IN    AL,DX
0006 A8 01                      TEST  AL,MASK1
0008 75 0F                      JNZ   LEVEL_0          ;if INTRB of first 8255A

000A A8 08                      TEST  AL,MASK2
000C 75 13                      JNZ   LEVEL_1          ;if INTRA of first 8255A

000E BA 0604                    MOV   DX,C2            ;address port C
0011 EC                         IN    AL,DX
0012 A8 01                      TEST  AL,MASK1
0014 75 1B                      JNZ   LEVEL_2          ;if INTRB of second 8255A

0016 EB 29 9                    JMP   LEVEL_3          ;if INTRA of second 8255A

0019                    POLL    ENDP
```

# 12–4 8259A PROGRAMMABLE INTERRUPT CONTROLLER

The 8259A programmable interrupt controller (PIC) adds eight vectored, priority-encoded interrupts to the microprocessor. This controller can be expanded, without additional hardware, to accept up to 64 interrupt request inputs. This expansion requires a master 8259A and eight 8259A slaves.

## General Description of the 8259A

Figure 12–15 shows the pinout of the 8259A. The 8259A is easy to connect to the microprocessor because all its pins are direct connections except the $\overline{CS}$ pin, which must be decoded, and the $\overline{WR}$ pin, which must have an I/O bank write pulse. Following is a description of each pin on the 8259A:

D7–D0 (Bidirectional Data Connections)—pins normally connected to either the upper or lower data bus on the 80386SX microprocessor, or the data bus on the 8088.

IR7–IR0 (Interrupt Request Inputs)—used to request an interrupt or to connect to a slave in a system with multiple 8259As.

$\overline{WR}$ (Write)—an input connected to either the lower or upper write strobe signal.

$\overline{RD}$ (Read)—an input connected to the $\overline{IORC}$ signal.

INT (Interrupt)—an output connected to the INTR pin on the microprocessor from the master, and to a master IR pin on a slave.

$\overline{INTA}$ (Interrupt Acknowledge)—an input connected to the $\overline{INTA}$ signal on the system. In a system with a master and slaves, only the master $\overline{INTA}$ signal is connected.

A0 (Address)—an input that selects different command words within the 8259A.

$\overline{CS}$ (Chip Select)—used to enable the 8259A for program and control.

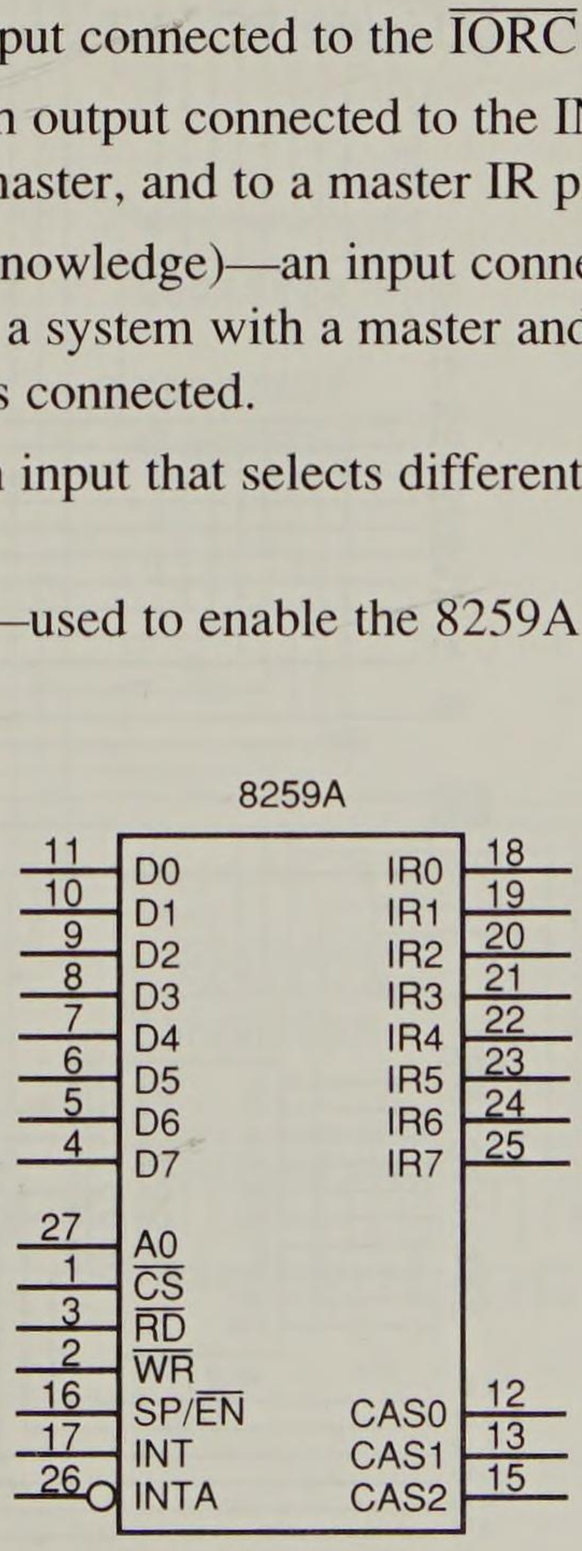



**FIGURE 12–15** The pinout of the 8259A programmable interrupt controller (PIC).

$\overline{\text{SP/EN}}$ (Slave Program/Enable Buffer)—a dual-function pin. When the 8259A is in buffered mode, this is an output that controls the data bus transceivers. When the 8259A is not in the buffered mode, this pin programs the device as a master (1) or a slave (0).

CAS2–CAS0 (Cascade Lines)—used as outputs from the master to the slaves for cascading multiple 8259As in a system.

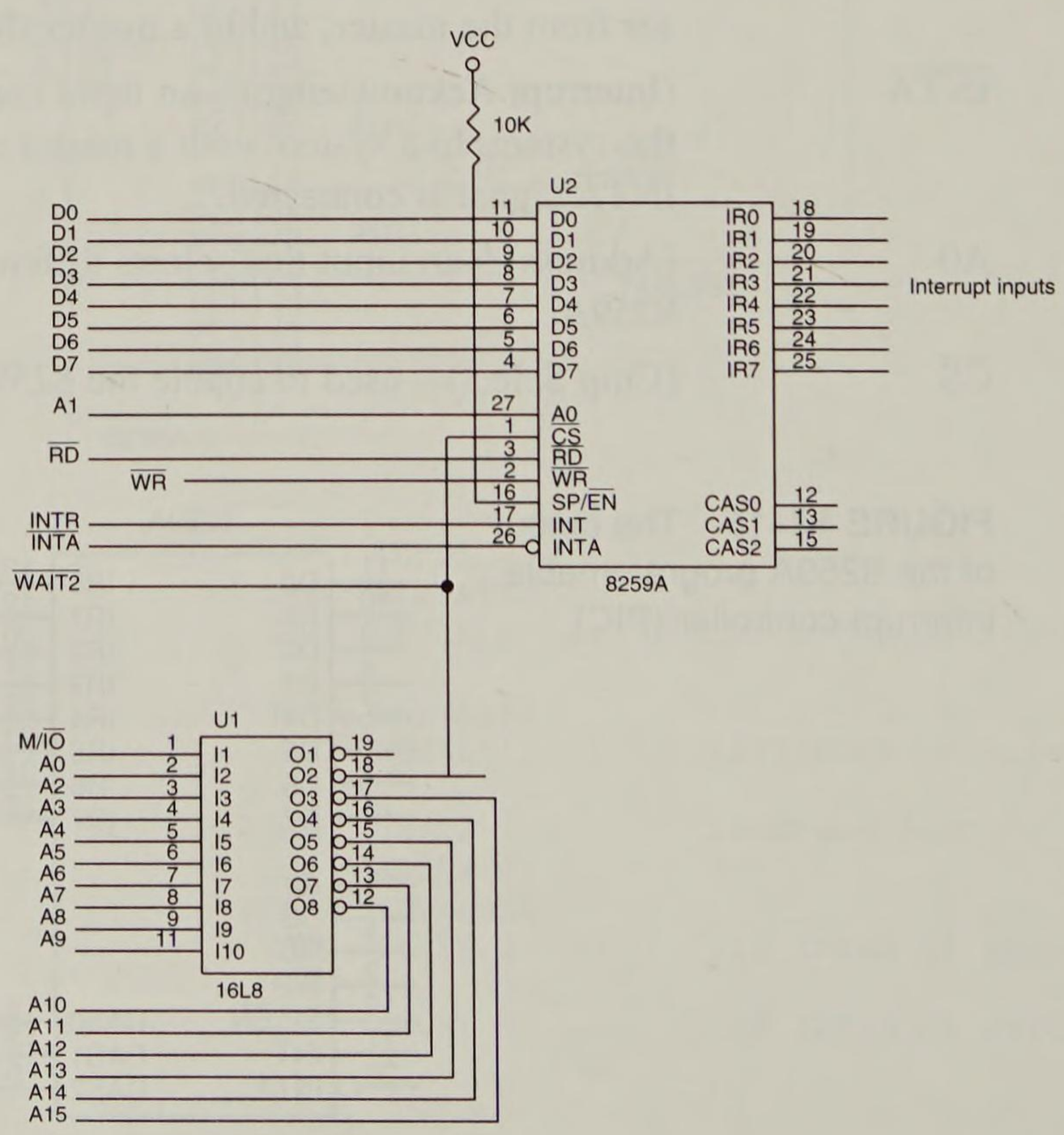
## Connecting a Single 8259A

Figure 12–16 shows a single 8259A connected to the 8086 microprocessor. Here, the $\overline{\text{SP/EN}}$ pin is pulled high to indicate that it is a master. Note that the 8259A is decoded at I/O ports 0400H and 0402H by the PAL16L8 (no program shown). Like other peripherals discussed in Chapter 11, the 8259A requires 4 wait states to function properly with a 16MHz 80386SX.

## Cascading Multiple 8259As

Figure 12–17 shows two 8259As connected to the 80386SX microprocessor in a way that is often found in the AT-style computer, which has two 8259As for interrupts. The XT- or PC-style computer uses one 8259A at interrupt vectors 08H through 0FH. The AT-style computer uses interrupt vector 0AH as a cascade input from a second 8259A located at vectors 70H through 77H. Appendix A contains a table that lists the functions of all the interrupt vectors used in the PC-, XT-, and AT-style computers.

**FIGURE 12–16** An 8259A interfaced to the 80X86 microprocessor.



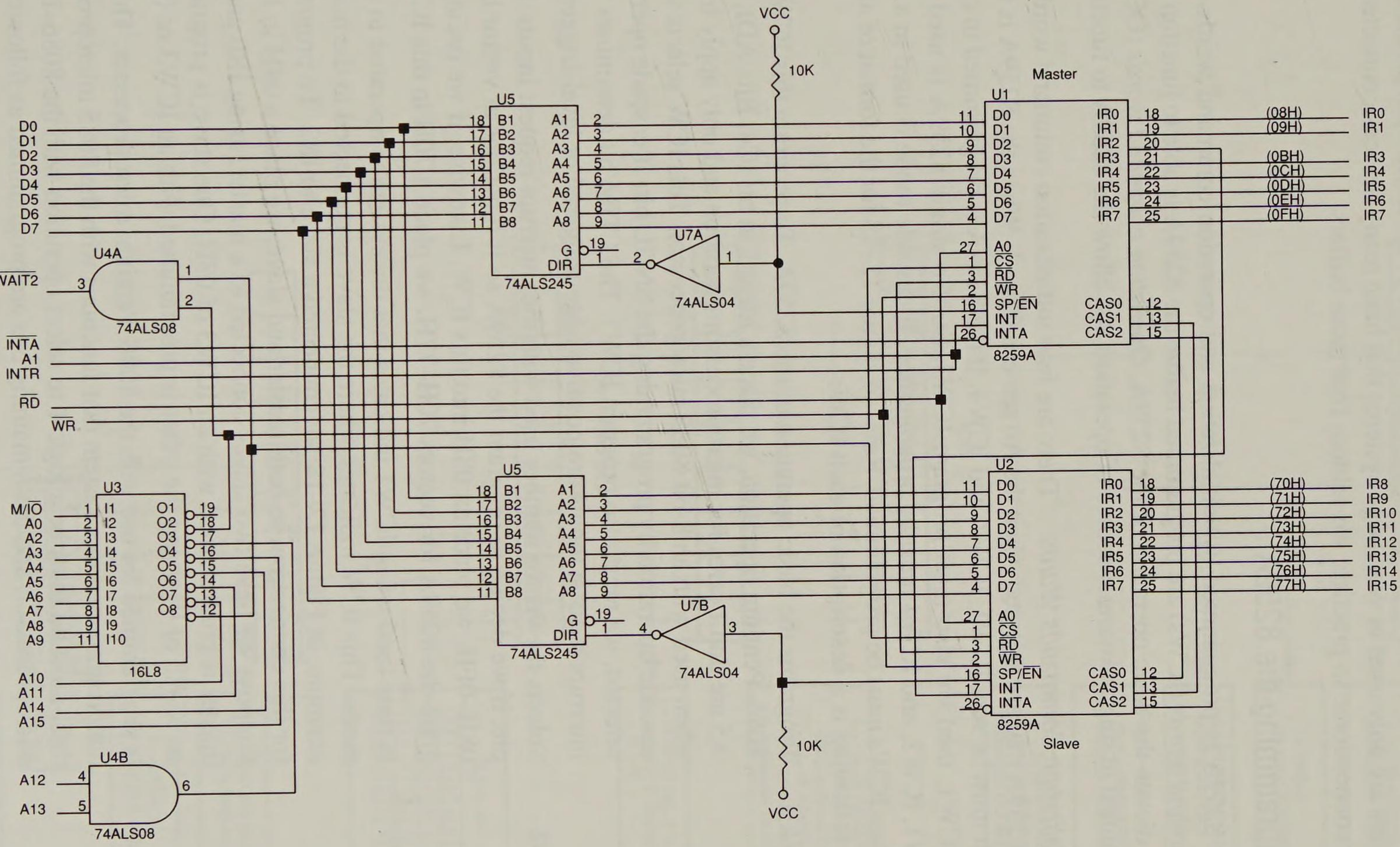



**FIGURE 12–17** Two 8259As interfaced to the 8259A at I/O ports 0300H and 0302H for the master and 0304H and 0306H for the slave.

This circuit uses vectors 08H–0FH and I/O ports 0300H and 0302H for U1, the master, and vectors 70H–77H and I/O ports 0304H and 0306H for U2, the slave. Notice that we also include data bus buffers to illustrate the use of the $\overline{SP/EN}$ pin on the 8259A. These buffers are only used in very large systems that have many devices connected to their data bus connections. In practice, we seldom find these buffers.

## Programming the 8259A

The 8259A is programmed by initialization and operation command words. *Initialization command words* (ICWs) are programmed before the 8259A is able to function in the system and dictate the basic operation of the 8259A. *Operation command words* (OCWs) are programmed during the normal course of operation and allow the 8259A to function properly.

*Initialization Commands Words.* There are four initialization command words (ICWs) for the 8259A that are selected when the A0 pin is a logic 1. When the 8259A is first powered up, it must be sent ICW1, ICW2, and ICW4. If the 8259A is programmed in cascade mode by ICW1, then we also must program ICW3. So if a single 8259A is used in a system, ICW1, ICW2, and ICW4 must be programmed. If cascade mode is used in a system, then all four ICWs must be programmed. Refer to Figure 12–18 for the format of all four ICWs. The following is a description of each ICW:

ICW1 Programs the basic operation of the 8259A. To program this ICW for 8086–Pentium operation, we place a logic 1 in bit IC4. Bits ADI, A7, A6, and A5 are don't cares for microprocessor operation and only apply to the 8259A when used with an 8-bit 8085 microprocessor. This ICW selects single or cascade operation by programming the SNGL bit. If cascade operation is selected, we must also program ICW3. The LTIM bit determines whether the interrupt request inputs are positive edge-triggered or level-triggered.

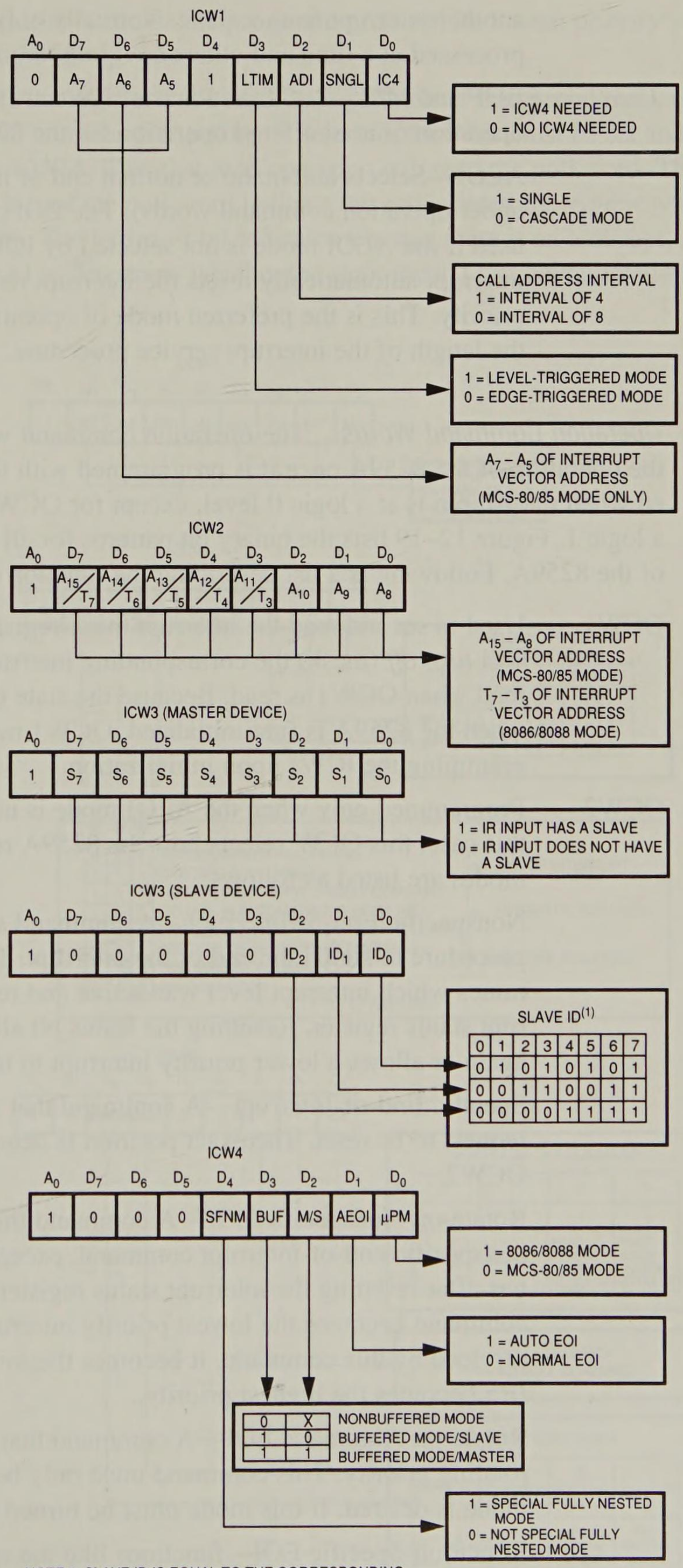
ICW2 Selects the vector number used with the interrupt request inputs. For example, if we decide to program the 8259A so it functions at vector locations 08H–0FH, we place an 08H into this ICW. Likewise, if we decide to program the 8259A for vectors 70H–77H, we place a 70H in this ICW.

ICW3 Is only used when ICW1 indicates that the system is operated in cascade mode. This ICW indicates where the slave is connected to the master. For example, in Figure 12–18 we connected a slave to IR2. To program ICW3 for this connection, in both master and slave, we place a 04H in ICW3. Suppose we have two slaves connected to a master, using IR0 and IR1. The master is programmed with an ICW3 of 03H. One slave is programmed with an ICW3 of 01H and the other is programmed with an ICW3 of 02H.

ICW4 Is programmed for use with the 8086–Pentium microprocessor. This ICW is not programmed in a system that functions with the 8085 microprocessor. The right-most bit must be a logic 1 to select operation with the 8086–Pentium microprocessor, and the remaining bits are programmed as follows:

SFNM—Selects the special fully nested mode of operation for the 8259A if a logic 1 is placed in this bit. This allows the highest priority interrupt request from a slave to be recognized by the master while it is processing

**FIGURE 12–18** The 8259A initialization command words (ICWs). (Courtesy of Intel Corporation)




NOTE 1: SLAVE ID IS EQUAL TO THE CORRESPONDING MASTER IR INPUT.

another interrupt from a slave. Normally only one interrupt request is processed at a time and others are ignored until the process is complete.

BUF and M/S—Buffer and master slave are used together to select buffered operation or nonbuffered operation for the 8259A as a master or a slave

AEOI—Selects automatic or normal end of interrupt (discussed more fully under operation command words). The EOI commands of OCW2 are only used if the AEOI mode is not selected by ICW4. If AEOI is selected, the interrupt automatically resets the interrupt request bit and does not modify priority. This is the preferred mode of operation for the 8259A and reduces the length of the interrupt service procedure.

*Operation Command Words.* The operation command words (OCWs) are used to direct the operation of the 8259A once it is programmed with the ICWs. The OCWs are selected when the A0 pin is at a logic 0 level, except for OCW1, which is selected when A0 is a logic 1. Figure 12–19 lists the binary bit patterns for all three operation command words of the 8259A. Following is a list describing the function of each OCW:

OCW1 Used to set and read the interrupt mask register. When a mask bit is set, it will *turn off* (mask) the corresponding interrupt input. The mask register is read when OCW1 is read. Because the state of the mask bits is unknown when the 8259A is first initialized, OCW1 must be programmed after programming the ICWs upon initialization.

OCW2 Programmed only when the AEOI mode is not selected for the 8259A. In this case, this OCW selects how the 8259A responds to an interrupt. The modes are listed as follows:

Nonspecific End-of-Interrupt—A command sent by the interrupt service procedure to signal the end of the interrupt. The 8259A automatically determines which interrupt level was active and resets the correct bit of the interrupt status register. Resetting the status bit allows the interrupt to take action again or allows a lower priority interrupt to take effect.

Specific End-of-Interrupt—A command that allows a specific interrupt request to be reset. The exact position is determined with bits L2–L0 of OCW2.

Rotate-on-Nonspecific EOI—A command that functions exactly like the nonspecific end-of-interrupt command, except that it rotates interrupt priorities after resetting the interrupt status register bit. The level reset by this command becomes the lowest priority interrupt. For example, if IR4 was just serviced by this command, it becomes the lowest priority interrupt input, and IR5 becomes the highest priority.

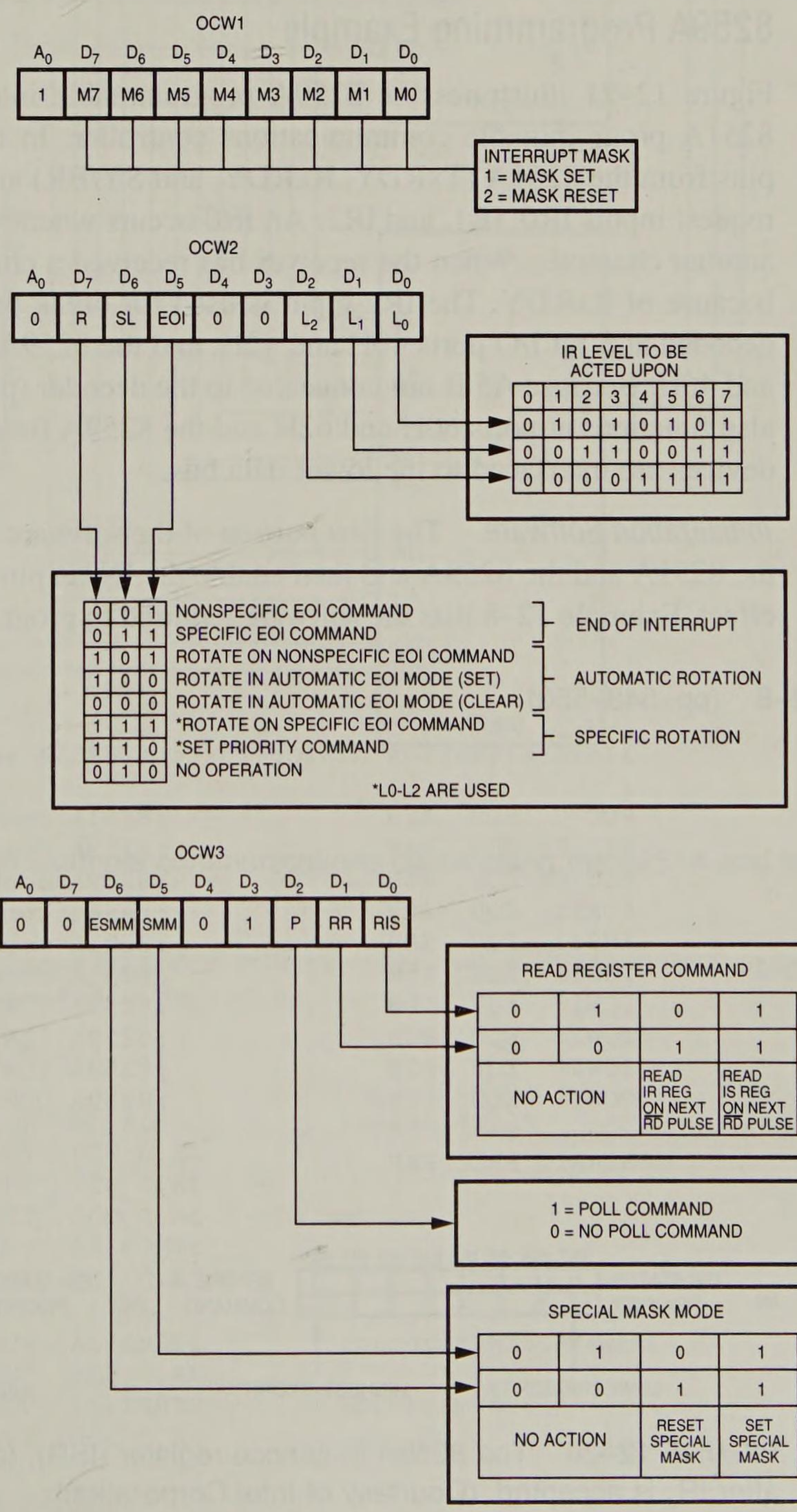
Rotate-on-Automatic EOI—A command that selects automatic EOI with rotating priority. This command must only be sent to the 8259A once, if this mode is desired. If this mode must be turned off, use the clear command.

Rotate-on-Specific EOI—functions like the specific EOI, except that it selects rotating priority.

Set priority—allows the programmer to set the lowest priority interrupt input using the L2–L0 bits.

OCW3 Selects the register to be read, the operation of the special mask register, and the poll command. If polling is selected, the P bit must be set and then output to the 8259A. The next read operation will read the poll word. The right-most three bits of the poll word indicate the active interrupt request with the highest priority. The left-most bit indicates whether there is an interrupt, and must be checked to determine whether the right-most 3 bits contain valid information.

**FIGURE 12–19** The 8259A operational command words (OCWs). (Courtesy of Intel Corporation)



*Status Register.* Three status registers are readable in the 8259A: interrupt request register (IRR), in-service register (ISR), and interrupt mask register (IMR). (Refer to Figure 12–20 for all three status registers.) The IRR is an 8-bit register that indicates which interrupt request inputs are active. The ISR is an 8-bit register that contains the level of the interrupt being serviced. The IMR is an 8-bit register that holds the interrupt mask bits and indicates which interrupts are masked off.

The IRR and ISR are read by programming OCW3, and the IMR is read through OCW1. To read the IMR, A0 = 1 and to read either IRR or ISR, A0 = 0. Bit positions D0 and D1 of OCW3 select which register (IRR or ISR) is read when A0 = 0.

## 8259A Programming Example

Figure 12–21 illustrates the 8259A programmable interrupt controller connected to an 8251A programmable communications controller. In this circuit, three interrupt output pins from the 8251A (TxRDY, RxRDY, and SY/BR) are connected to the PIC's interrupt request inputs IR0, IR1, and IR2. An IR0 occurs whenever the transmitter is ready to send another character. When the receiver has received a character for the 8086, IR1 is active because of RxRDY. The IR2 input is used for break detection. Notice that the 8251A is decoded at 8-bit I/O ports 40H and 42H, and the 8259A is decoded at 8-bit I/O ports 44H and 46H. Because A5 is not connected to the decoder (partially decoded ports), the 8251A also functions at ports 60H and 62H and the 8259A functions at ports 64H and 66H. Both devices are interfaced to the lower data bus.

*Initialization Software.* The first portion of the software for this system must program both the 8251A and the 8259A and then enable the INTR pin on the 8086 so interrupts can take effect. Example 12–8 lists the software required to program both devices and enable INTR.

**EXAMPLE 12–8** (pp. 548–550)

```
                        ;Initialization dialog for the 8251A and the 8259A
                        ;
= 0042                  PCC_C   EQU  42H             ;8251A control register
= 0044                  PIC_C1  EQU  44H             ;8259A control A0 = 0
= 0046                  PIC_C2  EQU  46H             ;8259A control A0 = 1
= 0040                  FORTY   EQU  40H             ;reset command for 8251A
= 007B                  INST1   EQU  7BH             ;8251A mode word
= 0015                  INST2   EQU  15H             ;8251A command word
= 001B                  ICW1    EQU  1BH             ;8259A ICW1
= 0080                  ICW2    EQU  80H             ;8259A ICW2
= 0003                  ICW4    EQU  03H             ;8259A ICW4
= 00F9                  OCW1    EQU  0F9H            ;8259A OCW1

0000                    START   PROC FAR
```

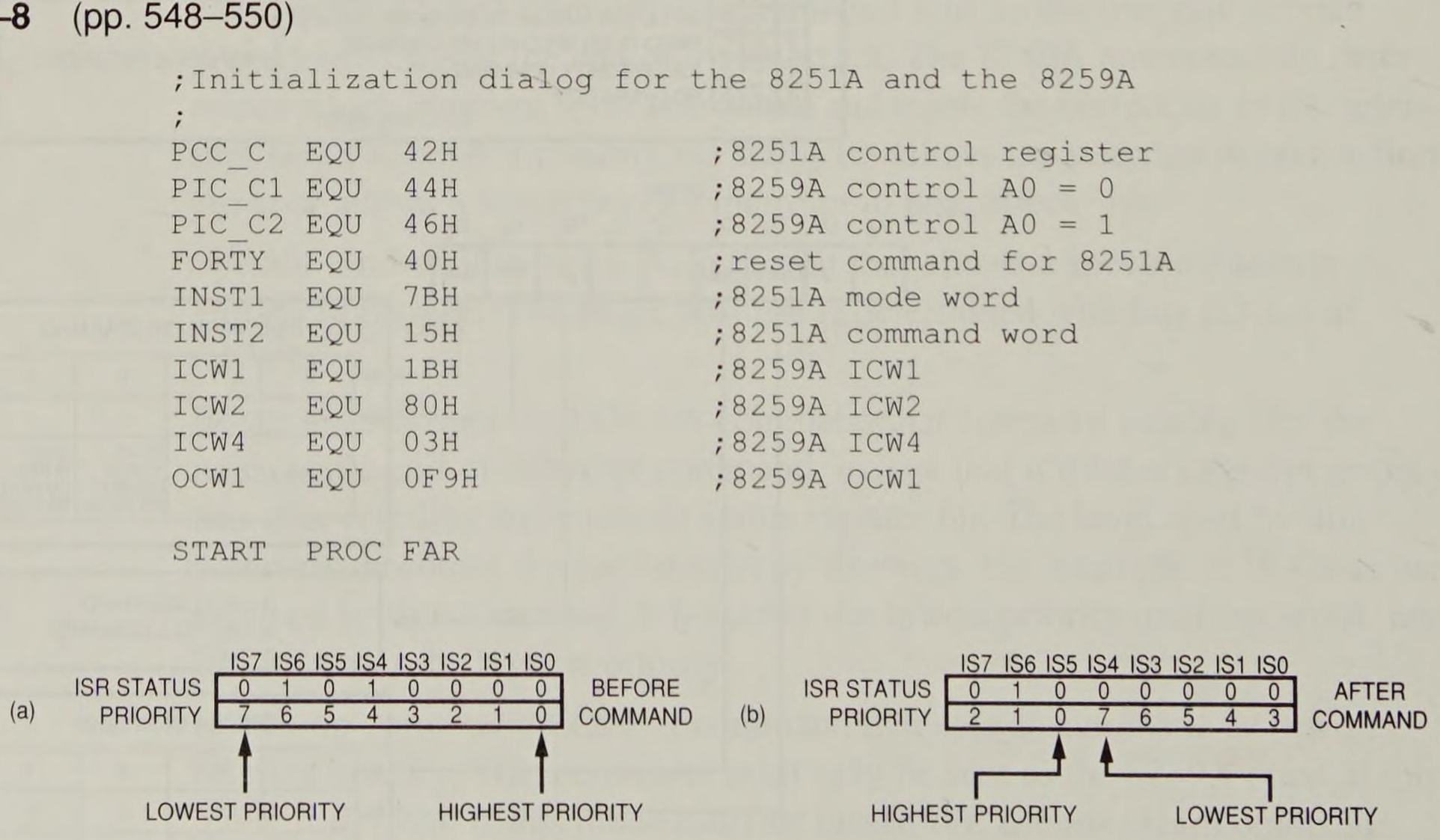



**FIGURE 12–20** The 8259A in-service register (ISR). (a) Before $IR_4$ is accepted, and (b) after $IR_4$ is accepted. (Courtesy of Intel Corporation)

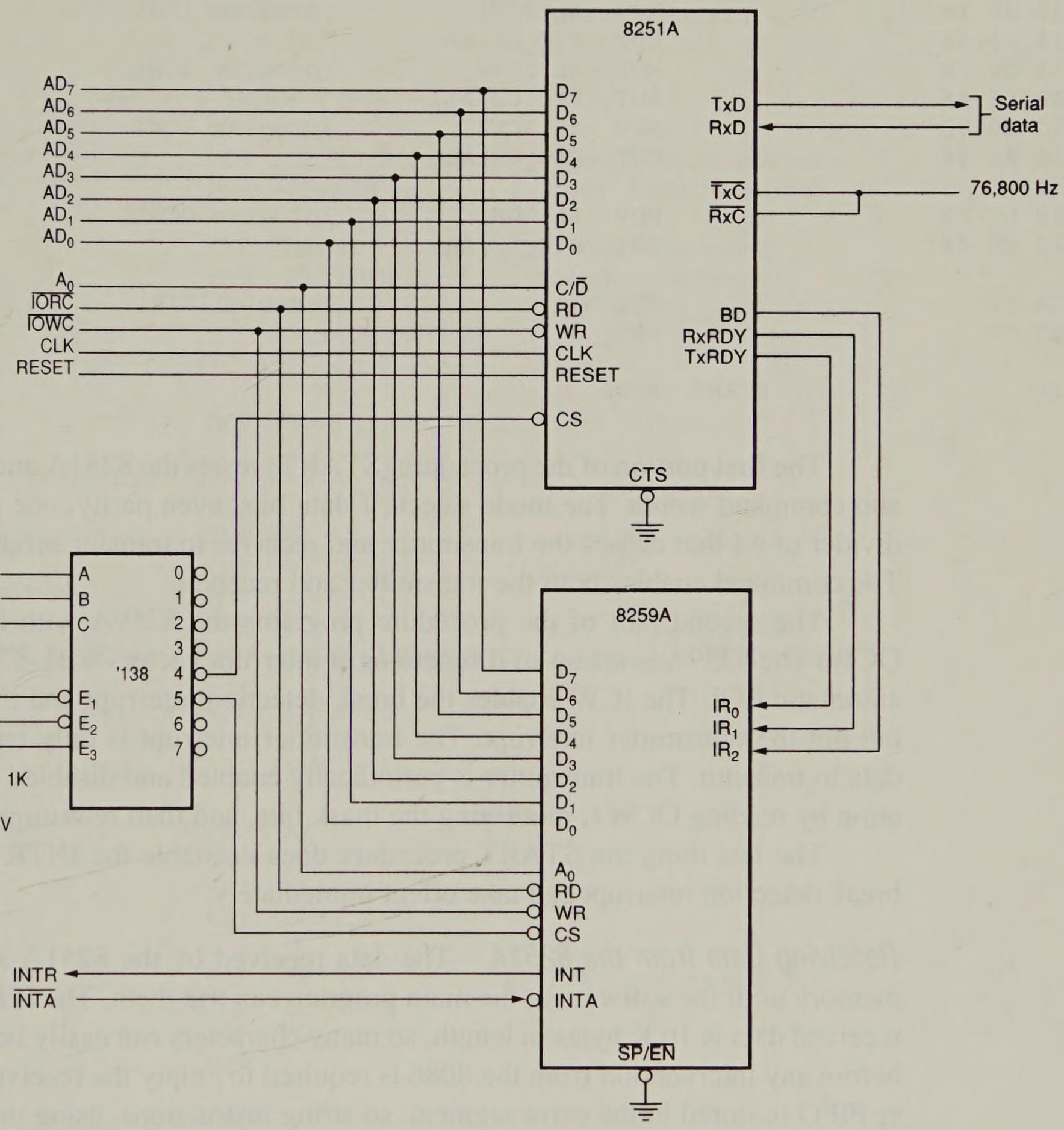



**FIGURE 12–21** An example communications circuit using the 8251A and the 8259A.

```
                ;set up the 8251A for 7 data bits, even parity, one stop bit,
                ;and a clock divider of 64.
                ;
                ;program 8251A
                ;
0000 32 C0              XOR  AL,AL          ;reset 8251A
0002 E6 42              OUT  PCC_C,AL
0004 E6 42              OUT  PCC_C,AL
0006 E6 42              OUT  PCC_C,AL
0008 B0 40              MOV  AL,FORTY
000A E6 42              OUT  PCC_C,AL

000C B0 7B              MOV  AL,INST1       ;program 8251A mode
000E E6 42              OUT  PCC_C,AL
0010 B0 15              MOV  AL,INST2       ;program 8251A command
0012 E6 42              OUT  PCC_C,AL

                ;program 8259A
```

```
0014 B0 1B                      MOV   AL,ICW1          ;program ICW1
0016 E6 44                      OUT   PIC_C1,AL
0018 B0 80                      MOV   AL,ICW2          ;program ICW2
001A E6 46                      OUT   PIC_C2,AL
001C B0 03                      MOV   AL,ICW4          ;program ICW4
001E E6 46                      OUT   PIC_C2,AL

0020 B0 F9                      MOV   AL,OCW1          ;program OCW1
0022 E6 46                      OUT   PIC_C2,AL

0024 FB                         STI                    ;enable INTR
0025 CB                         RET

0026                    START   ENDP
```

The first portion of the procedure (START) resets the 8251A and programs the mode and command words. The mode selects 7 data bits, even parity, one stop bit, and a clock divider of 64 that causes the transmitter and receiver to transfer serial data at 1200 Baud. The command enables both the transmitter and receiver.

The second part of the procedure programs the 8259A with three ICWs and one OCW. The 8259A is set up so it functions at interrupt vectors 80H–87H and operates with automatic EOI. The ICW enables the break detection interrupt and the receiver interrupt, but not the transmitter interrupt. The transmitter interrupt is only enabled when there is data to transmit. The transmitter is periodically enabled and disabled during normal operation by reading OCW1, modifying the mask bits, and then rewriting OCW1.

The last thing the START procedure does is enable the INTR pin so a receiver or break detection interrupt can take effect immediately.

*Receiving Data from the 8251A.* The data received by the 8251A are stored in a FIFO memory until the software in the main program can use them. The FIFO memory used for received data is 16 K bytes in length, so many characters can easily be stored and received before any intervention from the 8086 is required to empty the receiver FIFO. The receiver FIFO is stored in the extra segment, so string instructions, using the DI register, can be used to access it.

Receiving data from the 8251A requires two procedures: one reads the data register of the 8251A each time the RxRDY pin requests an interrupt and stores it into the FIFO, and the other reads data from the FIFO from the main program.

Example 12–9 lists the procedure used to read data from the FIFO from the main program. This procedure assumes that the pointers (IIN and IOUT) are initialized in the initialization dialog for the system (not shown). The READ procedure returns with AL containing a character read from the FIFO. If the FIFO is empty, the procedure returns with the character FEH in AL. This means that FEH is not allowed as a character at the receiver.

**EXAMPLE 12–9**

```
                        ;Main procedure to read a character from the FIFO
                        ;AL is the characater upon return or FEH if FIFO is empty.
                        ;
= 0046                  OCW1    EQU  46H                ;address of OCW1
= 00FD                  MASK1   EQU  0FDH

0000                    READ    PROC FAR
```

```
0000 53                         PUSH BX                 ;save registers
0001 57                         PUSH DI

                        ;test for empty FIFO

0002 26: 8B 3E 4002 R           MOV  DI,IOUT            ;get output pointer
0007 26: 8B 1E 4000 R           MOV  BX,IIN             ;get input pointer

000C 3B DF                      CMP  BX,DI              ;test for empty
000E B0 FE                      MOV  AL,0FEH            ;indicate empty
0010 74 16                      JE   DONE               ;if empty

                        ;get character from FIFO

0012 26: 8A 06                  MOV  AL,ES:[DI]         ;get data

                        ;modify and save output pointer

0015 47                         INC  DI                 ;adjust pointer
0016 81 FF 4000 R               CMP  DI,OFFSET FIFO+16*1024
001A 26: 89 3E 4002 R           MOV  IOUT,DI
001F 76 07                      JBE  DONE               ;if within bounds
0021 26: C7 06 4002 R 0000 R    MOV  IOUT,OFFSET FIFO ;point to FIFO start

                        ;enable RxRDY interrupt

0028                    DONE:

0028 50                         PUSH AX
0029 E4 46                      IN   AL,OCW1
002B 24 FD                      AND  AL,MASK1
002D E6 46                      OUT  OCW1,AL

002F 58                         POP  AX                 ;restore registers
0030 5F                         POP  DI
0031 5B                         POP  BX
0032 CB                         RET

0033                    READ    ENDP
```

Example 12–10 lists the RxRDY interrupt service procedure, which is called each time the 8251A receives a character for the 8086. In this example, this interrupt uses vector type number 81H, which must contain the address of the RxRDY interrupt service procedure. Each time this interrupt occurs, the RxRDY procedure reads a character from the 8251A and stores it into the FIFO. If the FIFO is full, the IR1 input to the 8259A is disabled. This may result in lost data, but at least it will not cause the interrupt to overrun valid data already stored in the FIFO. Any error conditions detected by the 8251A store a ? (3FH) in the FIFO.

**EXAMPLE 12–10**

```
                        ;Interrupt service procedure for RxRDY (IR1)
                        ;
= 0040                  DATA    EQU  40H                ;8251A data port
= 0046                  OCW1    EQU  46H                ;8259A OCW1 port
= 0002                  MASK2   EQU  02H                ;turn off IR1 mask
= 0042                  PCC_C   EQU  42H                ;8251A command port
= 0038                  MASK3   EQU  38H                ;8251A error mask

0000                    RXRDY   PROC FAR
```

```
0000 50                         PUSH AX                 ;save registers
0001 53                         PUSH BX
0002 57                         PUSH DI
0003 56                         PUSH SI
0004 26: 8B 1E 4002 R           MOV  BX,IOUT            ;load output pointer
0009 26: 8B 36 4000 R           MOV  SI,IIN             ;load input pointer

                        ;is FIFO full?

000E 8B FE                      MOV  DI,SI
0010 46                         INC  SI
0011 81 FE 4000 R               CMP  SI,OFFSET FIFO+16*1024
0015 76 03                      JBE  NEXT
0017 BE 0000 R                  MOV  SI,OFFSET FIFO

001A                    NEXT:

001A 3B DE                      CMP  BX,SI
001C 74 20                      JE   FULL

                        ;test for 8251A errors

001E E4 42                      IN   AL,PCC_C           ;get 8251A status register
0020 24 38                      AND  AL,MASK3
0022 74 0F                      JZ   NEXT1              ;if no errors
0024 B0 10                      MOV  AL,10H             ;reset errors
0026 E6 42                      OUT  PCC_C,AL
0028 B0 3F                      MOV  AL,'?'
002A AA                         STOSB
002B 26: 89 36 4000 R           MOV  IIN,SI
0030 EB 12 90                   JMP  DONE               ;if an error

                        ;read character

0033                    NEXT1:

0033 E4 40                      IN   AL,DATA            ;get 8251A data
0035 AA                         STOSB
0036 26: 89 36 4000 R           MOV  IIN,SI
003B EB 07 90                   JMP  DONE

                        ;if full
003E                    FULL:

003E E4 46                      IN   AL,OCW1            ;disable IR1
0040 0C 02                      OR   AL,MASK2
0042 E6 46                      OUT  OCW1,AL
0044                    DONE:

0044 5E                         POP  SI                 ;restore registers
0045 5F                         POP  DI
0046 5B                         POP  BX
0047 58                         POP  AX
0048 CF                         IRET

0049                    RXRDY   ENDP
```

*Transmitting Data to the 8251A.* Data are transmitted to the 8251A in much the same way they are received, except the interrupt service procedure removes transmit data from a second 16K byte FIFO.

Example 12–11 lists the procedure that fills the output FIFO. It is similar to the procedure listed in Example 12–9, except it determines whether the FIFO is full instead of empty.

**EXAMPLE 12–11**

```
                        ;Procedure that places data into the output FIFO for
                        ;transmission by the transmitter interrupt service procedure.
                        ;AL = character to be transmitted.
                        ;
= 0046                  OCW1    EQU  46H                ;8259A OCW1 port
= 00FE                  MASK4   EQU  0FEH               ;turn IR0 on

0000                    TRANS   PROC FAR

0000 53                         PUSH BX                 ;save registers
0001 57                         PUSH DI
0002 56                         PUSH SI

                        ;check if FIFO is full

0003 26: 8B 36 8004 R           MOV  SI,OIN             ;get input pointer
0008 26: 8B 1E 8006 R           MOV  BX,OOUT            ;get output pointer
000D 8B FE                      MOV  DI,SI
000F 46                         INC  SI
0010 81 FE 8004 R               CMP  SI,OFFSET OFIFO+16*1024
0014 76 03                      JBE  NEXT
0016 BE 4004 R                  MOV  SI,OFFSET OFIFO

0019                    NEXT:

0019 3B DE                      CMP  BX,SI
001B 74 06                      JE   DONE               ;if full
001D AA                         STOSB
001E 26: 89 36 8004 R           MOV  OIN,SI

                        DONE:

0023 E4 46                      IN   AL,OCW1            ;enable transmitter interrupt
0025 24 FE                      AND  AL,MASK4
0027 E6 46                      OUT  OCW1,AL

0029 5E                         POP  SI                 ;restore registers
002A 5F                         POP  DI
002B 5B                         POP  BX
002C CB                         RET

002D                    TRANS   ENDP
```

Example 12–12 lists the interrupt service subroutine for the 8251A transmitter, TxRDY, using interrupt vector number 80H. This procedure is similar to the RxRDY procedure of Example 12–10, except it determines whether the FIFO is empty rather than full. Note that we do not include an interrupt service procedure for the break interrupt.

**EXAMPLE 12–12**

```
                        ;Interrupt service procedure for the 8251A transmitter
                        ;
= 0040                  DATA    EQU  40H                ;8251A data port
```

```
= 0001                  MASK5  EQU  01H              ;turn IR0 off

002D                    TXRDY  PROC FAR

002D 50                        PUSH AX              ;save registers
002E 53                        PUSH BX
002F 57                        PUSH DI
0030 26: 8B 1E 8004 R          MOV  BX,OIN          ;load input pointer
0035 26: 8B 3E 8006 R          MOV  DI,OOUT         ;load output pointer

                        ;is FIFO empty?

003A 3B DF                     CMP  BX,DI
003C 74 17                     JE   EMPTY           ;if empty

                       ;write character

003E 26: 8A 05                 MOV  AL,ES:[DI]
0041 E6 40                     OUT  DATA,AL
0043 47                        INC  DI
0044 81 FF 8004 R              CMP  DI,OFFSET OFIFO+16*1024
0048 76 03                     JBE  NEXT1
004A BF 4004 R                 MOV  DI,OFFSET OFIFO

004D                    NEXT1:

004D 26: 89 3E 8006 R          MOV  OOUT,DI
0052 EB 07 90                  JMP  DONES

0055                    EMPTY:

0055 E4 46                     IN   AL,OCW1
0057 0C 01                     OR   AL,MASK5
0059 E6 46                     OUT  OCW1,AL

005B                    DONE:

005B 5F                        POP  DI              ;restore registers
005C 5B                        POP  BX
005D 58                        POP  AX
005E CF                        IRET

005F                    TXRDY  ENDP
```

## 12–5 REAL-TIME CLOCK

This section of the text presents a real-time clock as an example use of an interrupt. A real-time clock keeps time in real time; that is, in hours and minutes. The example illustrated here keeps time in hours, minutes, seconds, and 1/60 seconds, using four memory locations to hold the BCD time of day.

Figure 12–22 illustrates a simple circuit that uses the 60 Hz AC power line to generate a periodic interrupt request signal for the NMI interrupt input pin. We are using a signal from the AC power line, which, although it varies slightly in frequency from time to time, is accurate over a period of time. The circuit uses a signal from the 120 VAC power

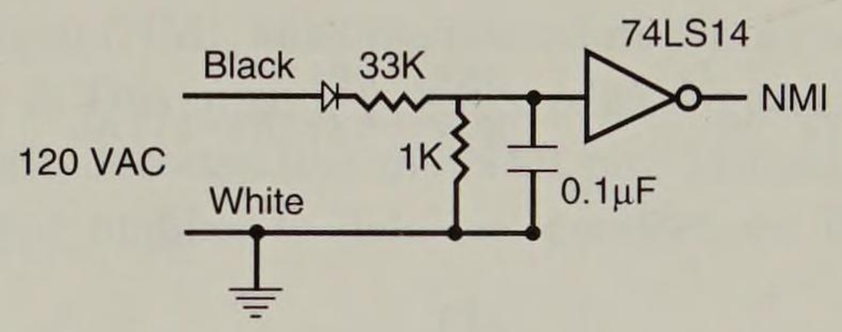



**FIGURE 12–22** Converting the AC power line to a 60-Hz TTL signal for the NMI input.

line that is conditioned by a Schmitt-trigger inverter before it is applied to the NMI interrupt input. You must make certain that the power line ground is connected to the system ground in this schematic. The power line ground (neutral) connection is the large flat pin on the power line. The narrow flat pin is the hot side or 120 VAC side of the line.

The software for the real-time clock contains an interrupt service procedure that is called 60 times per second, and a procedure that updates the count located in four memory locations. Example 12–13 lists both procedures along with the four bytes of memory used to hold the BCD time of day.

**EXAMPLE 12–13**

```
0000 00                 TIME    DB    ?               ;1/60 seconds counter
0001 00                         DB    ?               ;seconds counter
0002 00                         DB    ?               ;minutes counter
0003 00                         DB    ?               ;hours counter (24 hour clock)

                        ;Interrupt service procedure for NMI
0004                    TIMES   PROC FAR

0004 50                         PUSH AX               ;save registers
0005 56                         PUSH SI

0006 B4 60                      MOV  AH,60H           ;load modulus of counter
0008 BE 0000 R                  MOV  SI,OFFSET TIME   ;address time
000B E8 0022 R                  CALL UP               ;adjust 1/60 counter
000E 75 0F                      JNZ  DONE
0010 E8 0022 R                  CALL UP               ;adjust seconds counter
0013 75 0A                      JZ   DONE
0015 E8 0022 R                  CALL UP               ;adjust minutes
0018 75 05                      JZ   DONE
001A B4 24                      MOV  AH,24H           ;modulus 24
001C E8 0022 R                  CALL UP

001F                    DONE:

001F 5E                         POP  SI               ;restore registers
0020 58                         POP  AX
0021 CF                         IRET

0022                    TIMES   ENDP

0022                    UP      PROC NEAR

0022 2E: 8A 04                  MOV  AL,CS:[SI]
0025 46                         INC  SI
0026 04 01                      ADD  AL,1             ;increment counter
0028 27                         DAA
0029 2E: 88 44 FF               MOV  CS:[SI-1],AL
```

```
002D 2A C4                      SUB   AL,AH
002F 75 04                      JNZ   UP1
0031 2E: 88 44 FF               MOV   CS:[SI-1],AL

0035                    UP1:

0035C3                          RET

0036                    UP      ENDP
```

## 12–6 SUMMARY

1. An interrupt is a hardware- or software-initiated call that interrupts the currently executing program at any point and calls a procedure. The procedure called by the interrupt is an interrupt service procedure.
2. Interrupts are useful when an I/O device needs to be serviced only occasionally at low data transfer rates.
3. The microprocessor has 5 instructions that apply to interrupts: BOUND, INT, INT 3, INTO, and IRET. The INT and INT 3 instructions call procedures with addresses stored in an interrupt vector, the type of which is indicated by the instruction. The BOUND instruction is a conditional interrupt that uses interrupt vector type number 5. The INTO instruction is a conditional interrupt that only interrupts a program if the overflow flag is set. Finally, the IRET instruction is used to return from interrupt service procedures.
4. The microprocessor has three pins that apply to its hardware interrupt structure: INTR, NMI, and $\overline{\text{INTA}}$. INTR and NMI are interrupt inputs that are used to request interrupts, and $\overline{\text{INTA}}$ is an output used to acknowledge the INTR interrupt request.
5. Real-mode interrupts are referenced through a vector table, which occupies memory locations 00000H–003FFH. Each interrupt vector is four bytes in length and contains the offset and segment addresses of the interrupt service procedure. In protected mode, the interrupts reference the interrupt descriptor table (IDT), which contains 256 interrupt descriptors. Each interrupt descriptor contains a segment selector and a 32-bit offset address.
6. Two flag bits are used with the interrupt structure of the microprocessor: trap (TF) and interrupt enable (IF). The IF flag bit enables the INTR interrupt input and the TF flag bit causes interrupts to occur after the execution of each instruction, as long as TF is active.
7. The first 32 interrupt vector locations are reserved for Intel use, with many predefined in the microprocessor. The last 224 interrupt vectors are for user use and can perform any function desired.
8. Whenever an interrupt is detected, the following events occur: (1) the flags are pushed onto the stack, (2) the IF and TF flag bits are both cleared, (3) the IP and CS registers are both pushed onto the stack, and (4) the interrupt vector is fetched from the interrupt vector table and the interrupt service subroutine is accessed through the vector address.
9. Tracing or single-stepping is accomplished by setting the TF flag bit. This causes an interrupt to occur after the execution of each instruction for debugging.

10. The nonmaskable interrupt input (NMI) calls the procedure whose address is stored at interrupt vector type number 2. This input is positive edge-triggered.
11. The INTR pin is not internally decoded like the NMI pin. Instead, $\overline{\text{INTA}}$ is used to apply the interrupt vector type number to data bus connections D0–D7 during the $\overline{\text{INTA}}$ pulse.
12. Methods of applying the interrupt vector type number to the data bus during $\overline{\text{INTA}}$ vary widely. One method uses resisters to apply interrupt type number FFH to the data bus, while another uses a three-state buffer to apply any vector type number.
13. The 8259A programmable interrupt controller (PIC) adds at least eight interrupt inputs to the microprocessor. If more interrupts are needed, this device can be cascaded to provide up to 64 interrupt inputs.
14. Programming the 8259A is a two-step process. First, a series of initialization command words (ICWs) are sent to the 8259A, then a series of operation command words (OCWs) are sent.
15. The 8259A contains three status registers: IMR (interrupt mask register), ISR (in-service register), and IRR (interrupt request register).
16. A real-time clock is used to keep time in real time. In most cases, time is stored in either binary or BCD form in several memory locations.

## 12–7 QUESTIONS AND PROBLEMS

1. What is interrupted by an interrupt?
2. Define the term *interrupt*.
3. What is called by an interrupt?
4. Why do interrupts free-up time for the microprocessor?
5. List the interrupt pins found on the microprocessor.
6. List the 5 interrupt instructions for the microprocessor.
7. What is an interrupt vector?
8. Where are the interrupt vectors located in the microprocessor's memory?
9. How many different interrupt vectors are found in the interrupt vector table?
10. Which interrupt vectors are reserved by Intel?
11. Explain how a type 0 interrupt occurs.
12. Where is the interrupt descriptor table located for protected-mode operation?
13. Each protected-mode interrupt descriptor contains what information?
14. Describe the differences between a protected- and real-mode interrupt.
15. Describe the operation of the BOUND instruction.
16. Describe the operation of the INTO instruction.
17. What memory locations contain the vector for an INT 44H instruction?
18. Explain the operation of the IRET instruction.
19. What is the purpose of interrupt vector type number 7?
20. List the events that occur when an interrupt becomes active.
21. Explain the purpose of the interrupt flag (IF).
22. Explain the purpose of the trap flag (TF).
23. How is IF cleared and set?

24. How is TF cleared and set?
25. The NMI interrupt input automatically vectors through which vector type number?
26. Does the $\overline{\text{INTA}}$ signal activate for the NMI pin?
27. The INTR input is _____ -sensitive.
28. The NMI input is _____ -sensitive.
29. When the $\overline{\text{INTA}}$ signal becomes a logic 0, it indicates that the microprocessor is waiting for an interrupt _____ number to be placed on the data bus (D0–D7).
30. What is a FIFO?
31. Develop a circuit that places interrupt type number 86H on the data bus in response to the INTR input.
32. Develop a circuit that places interrupt type number CCH on the data bus in response to the INTR input.
33. Explain why pullup resistors on D0–D7 cause the microprocessor to respond with interrupt vector type number FFH for the $\overline{\text{INTA}}$ pulse.
34. What is a daisy chain?
35. Why must interrupting devices be polled in a daisy-chained interrupt system?
36. What is the 8259A?
37. How many 8259As are required to have 64 interrupt inputs?
38. What is the purpose of the IR0–IR7 pins on the 8259A?
39. When are the CAS2–CAS0 pins used on the 8259A?
40. Where is a slave INT pin connected on the master 8259A in a cascaded system?
41. What is an ICW?
42. What is an OCW?
43. How many ICWs are needed to program the 8259A when operated as a single master in a system?
44. Where is the vector type number stored in the 8259A?
45. Where is the sensitivity of the IR pins programmed in the 8259A?
46. What is the purpose of ICW1?
47. What is a nonspecific EOI?
48. Explain priority rotation in the 8259A.
49. What is the purpose of IRR in the 8259A?
50. At which I/O ports is the master 8259A PIC found in the personal computer?
51. At which I/O ports is the slave 8259A found in the personal computer?

# CHAPTER 13

# Direct Memory Access and DMA-Controlled I/O

## INTRODUCTION

In previous chapters, we discussed basic and interrupt-processed I/O. Now we turn to the final form of I/O called *direct memory access* (DMA). The DMA I/O technique provides direct access to the memory while the microprocessor is temporarily disabled. This allows data to be transferred between memory and the I/O device at a rate that is limited only by the speed of the memory components in the system or the DMA controller. The DMA transfer speed can approach 10–12 M byte transfer rates with today's high-speed RAM memory components.

DMA transfers are used for many purposes, but most common are DRAM refresh, video displays for refreshing the screen, and disk memory system reads and writes. The DMA transfer is also used to do high-speed memory-to-memory transfers.

This chapter also explains the operation of disk memory systems and video systems, which are often DMA processed. Disk memory includes: floppy, fixed, and optical disk storage. Video systems include: digital and analog monitors.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Describe a DMA transfer.
2. Explain the operation of the HOLD and HLDA direct memory access control signals.
3. Explain the function of the 8237 DMA controller when used for DMA transfers.
4. Program the 8237 to accomplish DMA transfers.
4. Describe the disk standards found in personal computer systems.
5. Describe the various video interface standards found in the personal computer.

## 13–1 BASIC DMA OPERATION

Two control signals are used to request and acknowledge a direct memory access (DMA) transfer in the microprocessor-based system. The HOLD pin is an input used to request a DMA action and the HLDA pin is an output that acknowledges the DMA action. Figure 13–1 shows the timing typically found on these two DMA control pins.

Whenever the HOLD input is placed at a logic 1 level, a DMA action (hold) is requested. The microprocessor responds, within a few clocks, by suspending the execution of the program and by placing its address, data, and control bus at their high-impedance states. The high-impedance state causes the microprocessor to appear as if it has been removed from its socket. This state allows external I/O devices or other microprocessors to gain access to the system buses, so memory can be accessed directly.

As the timing diagram indicates, HOLD is sampled in the middle of any clocking cycle. Thus, the hold can take effect at any time during the operation of any instruction in the microprocessor. As soon as the microprocessor recognizes the hold, it stops executing software and enters hold cycles. Note that the HOLD input has a higher priority than the INTR or NMI interrupt inputs. Interrupts take effect at the end of an instruction, while a HOLD takes effect in the middle of an instruction. The only microprocessor pin that has a higher priority than a HOLD is the RESET pin. Note that the HOLD input may not be active during a RESET, or the reset is not guaranteed.

The HLDA signal becomes active to indicate that the microprocessor has indeed placed its buses at their high-impedance state, as can be seen in the timing diagram. Note that there are a few clock cycles between the time the HOLD changes and the time the HLDA changes. The HLDA output is a signal to the external requesting device that the microprocessor has relinquished control of its memory and I/O space. You could call the HOLD input a DMA request input, and the HLDA output a DMA grant signal.

### Basic DMA Definitions

Direct memory accesses normally occur between an I/O device and memory without the use of the microprocessor. A *DMA read* transfers data from the memory to the I/O device. A *DMA write* transfers data from an I/O device to memory. In both operations, the memory and I/O are controlled simultaneously, which is why the system contains separate memory and I/O control signals. This special control bus structure of the microprocessor allows DMA transfers. A DMA read causes the $\overline{\text{MRDC}}$ ($\overline{\text{MEMR}}$) and $\overline{\text{IOWC}}$ ($\overline{\text{IOW}}$) signals to both activate, transferring data from the memory to the I/O device. A

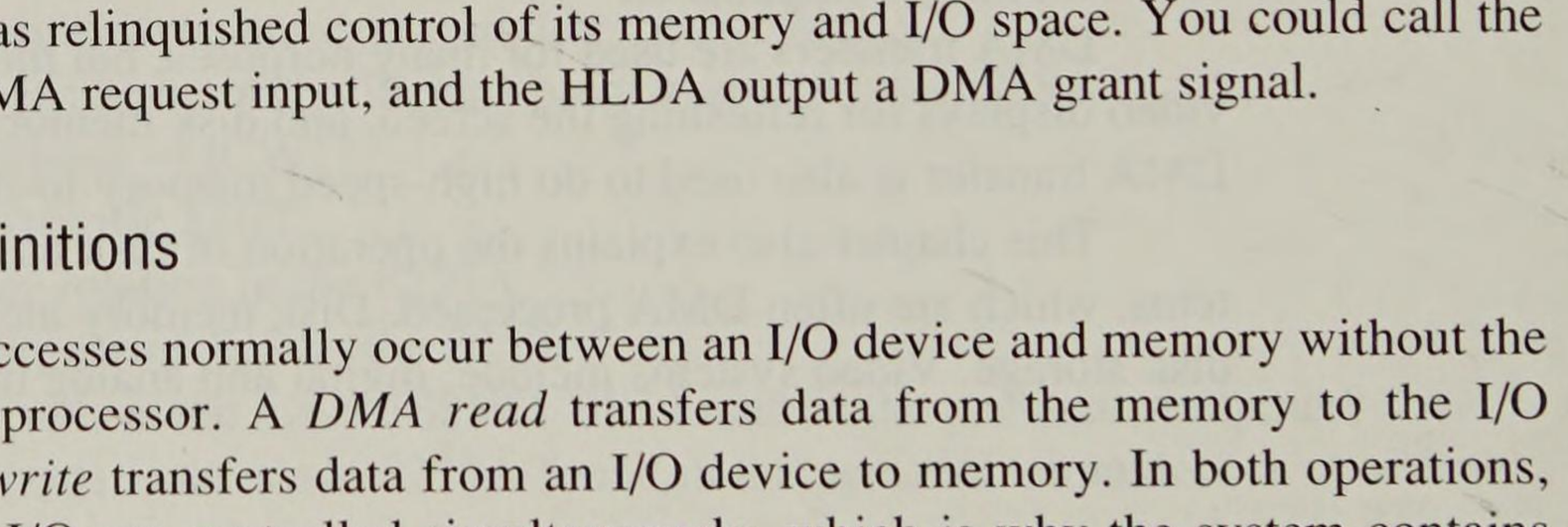



**FIGURE 13–1** HOLD and HLDA timing for the 8086/8088 microprocessor.

DMA write causes the $\overline{\text{MWTC}}$ ($\overline{\text{MEMW}}$) and $\overline{\text{IORC}}$ ($\overline{\text{IOR}}$) signals to both activate. These control bus signals are available to all microprocessors in the Intel family except the 8086/8088 system. The control signals require generation with either a system controller or a circuit, like the one illustrated in Figure 13–2. The DMA controller provides the memory with its address, and a signal from the controller ($\overline{\text{DACK}}$) selects the I/O device during the DMA transfer.

The data transfer speed is determined by the speed of the memory device or a DMA controller. If the memory speed is 100 ns, DMA transfers occur at rates of up to 1/100 ns or 10M bytes per second. If the DMA controller in a system functions at a maximum rate of 5 MHz, and we still use 100 ns memory, the maximum transfer rate is 5 MHz, because the DMA controller is slower than the memory. In many cases the DMA controller slows the speed of the system when DMA transfers occur.

## 13–2 THE 8237 DMA CONTROLLER

The 8237 DMA controller supplies the memory and I/O with control signals and memory address information during the DMA transfer. The 8237 is actually a special purpose microprocessor whose job is high-speed data transfer between memory and the I/O. Figure 13–3 shows the pinout and block diagram of the 8237 programmable DMA controller. This device may not appear as a discrete component in modern microprocessor-based systems, but it does appear within system controller chip sets found in most newer systems. Although not described because of its complexity, the chip set (82357 ISP or integrated peripheral controller) and its integral set of two DMA controllers are programmed exactly like the 8237. The ISP also provides a pair of 8259A programmable interrupt controllers for the system.

The 8237 is a four-channel device which is compatible to the 8086/8088 microprocessor. The 8237 can be expanded to include any number of DMA channel inputs, although four channels seem to be adequate for many small systems. The 8237 is capable

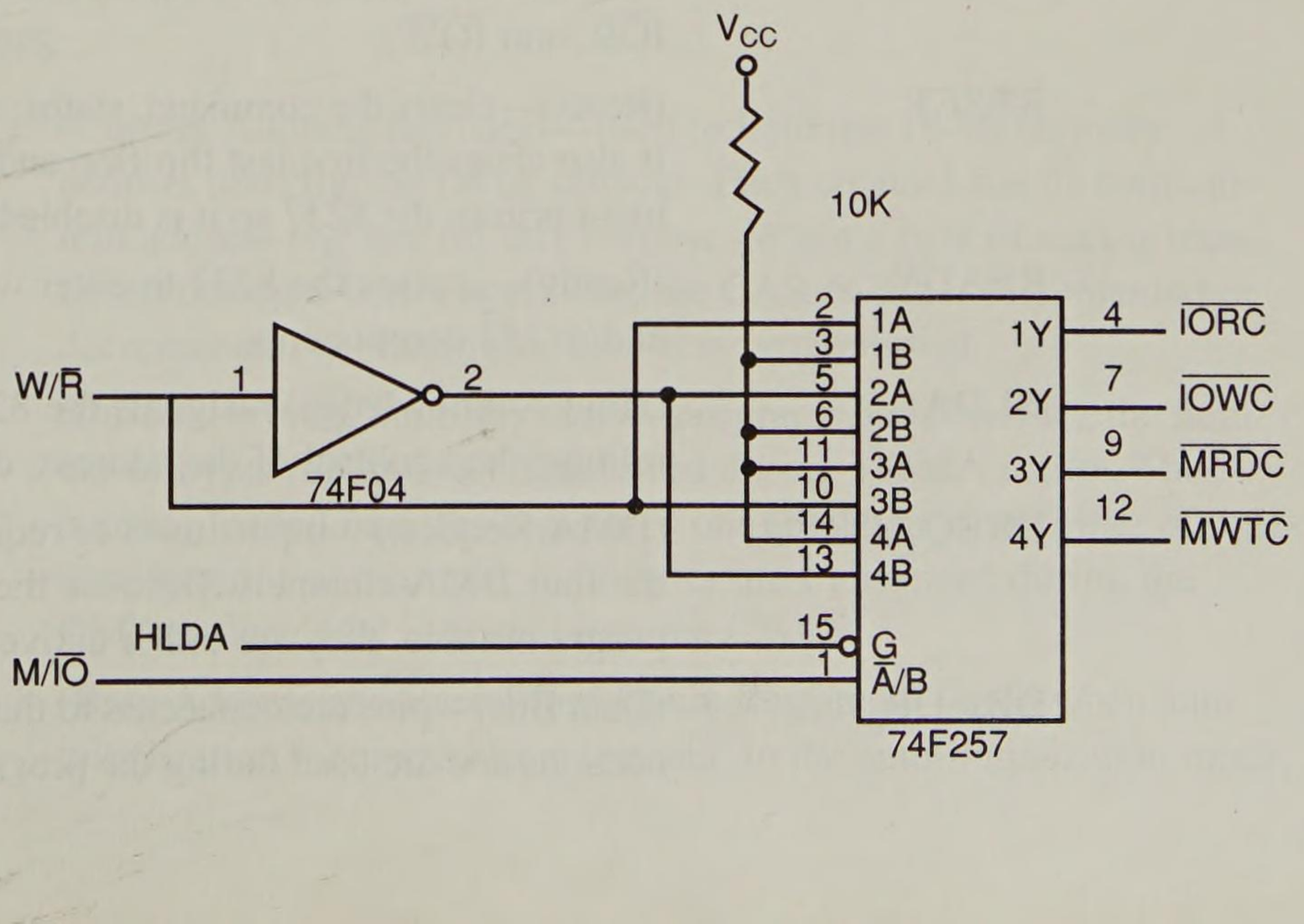



**FIGURE 13–2** A circuit that generates system control signals in a DMA environment.

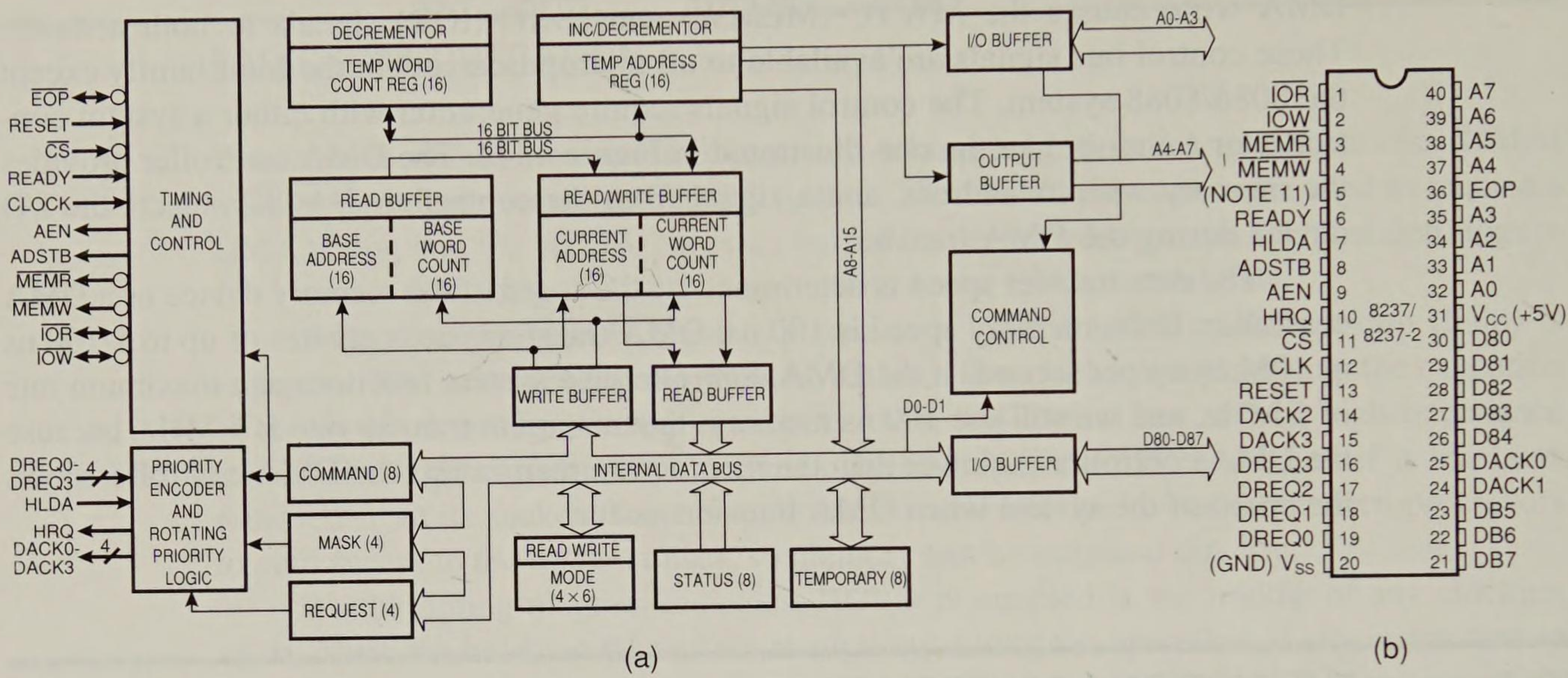



**FIGURE 13–3** The 8237A-5 programmable DMA controller. (a) Block diagram, and (b) pinout. (Courtesy of Intel Corporation)

of DMA transfers at rates of up to 1.6M bytes per second. Each channel is capable of addressing a full 64K-byte section of memory, and can transfer up to 64K bytes with a single programming.

## Pin Definitions

| | |
|---|---|
| CLK | (Clock)—is connected to the system clock signal as long as that signal is 5 MHz or less. In the 8086/8088 system, the clock must be inverted for proper operation of the 8237. |
| $\overline{CS}$ | (Chip Select)—selects the 8237 during programming. The CS pin is normally connected to the output of a decoder. The decoder does not use the 8086/8088 control signal IO/$\overline{M}$ (M/$\overline{IO}$) because it contains the new memory and I/O control signals ($\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$, and $\overline{IOW}$). |
| RESET | (Reset)—clears the command, status, request, and temporary registers. It also clears the first/last flip-flop and sets the mask register. This input primes the 8237 so it is disabled until programmed otherwise. |
| READY | (Ready)—causes the 8237 to enter wait states for slower memory and/or I/O components. |
| HLDA | (Hold Acknowledge)—signals the 8237 that the microprocessor has relinquished control of the address, data, and control buses. |
| $DREQ_3$–$DREQ_0$ | (DMA Request)—inputs used to request a DMA transfer for each of the four DMA channels. Because the polarity of these inputs is programmable, they are either active-high or active-low inputs. |
| $DB_7$–$DB_0$ | (Data Bus)—pins are connected to the microprocessor data bus connections and are used during the programming of the DMA controller. |

$\overline{IOR}$ (I/O Read)—a bidirectional pin used during programming and during a DMA read cycle.

$\overline{IOW}$ (I/O Write)—a bidirectional pin used during programming and during a DMA write cycle.

$\overline{EOP}$ (End-of-Process)—a bidirectional signal used as an input to terminate a DMA process, or as an output to signal the end of the DMA transfer. This input is often used to interrupt a DMA transfer at the end of a DMA cycle.

$A_3$–$A_0$ (Address)—pins used to select an internal register during programming and to provide part of the DMA transfer address during a DMA action.

$A_7$–$A_4$ (Address)—outputs that provide part of the DMA transfer address during a DMA action.

HRQ (Hold Request)—an output that connects to the HOLD input of the microprocessor in order to request a DMA transfer.

$DACK_3$–$DACK_0$ (DMA Channel Acknowledge)—outputs that acknowledge a channel DMA request. These outputs are programmable as either active-high or active-low signals. The DACK outputs are often used to select the DMA-controlled I/O device during the DMA transfer.

AEN (Address Enable)—enables the DMA address latch connected to the $DB_7$–$DB_0$ pins on the 8237. It is also used to disable any buffers in the system connected to the microprocessor.

ADSTB (Address Strobe)—functions like ALE, except it is used by the DMA controller to latch address bits $A_{15}$–$A_8$ during the DMA transfer.

$\overline{MEMR}$ (Memory Read)—an output that causes memory to read data during a DMA read cycle.

$\overline{MEMW}$ (Memory Write)—an output that causes memory to write data during a DMA write cycle.

## Internal Registers

CAR (Current Address Register)—used to hold the 16-bit memory address used for the DMA transfer. Each channel has its own current address register for this purpose. When a byte of data is transferred during a DMA operation, the CAR is either incremented or decremented, depending on how it is programmed.

CWCR (Current Word Count Register)—programs a channel for the number of bytes (up to 64K) transferred during a DMA action. The number loaded into this register is one less than the number of bytes transferred. For example, if 11 bytes are transferred during the DMA action, a 10 is loaded into the CWCR.

BA and BWC (Base Address and Base Word Count Registers)—used when auto-initialization is selected for a channel. In the auto-initialization mode,

these registers are used to reload both the CAR and the CWCR after the DMA action is completed. This allows the same count and address to be used to transfer data from the same memory area.

CR (Command register)—programs the operation of the 8237 DMA controller. Figure 13–4 depicts the function of the command register.

The command register uses bit position 0 to select the memory-to-memory DMA transfer mode. Memory-to-memory DMA transfers use DMA channel 0 to hold the source address, and DMA channel 1 to hold the destination address. (This is similar to the operation of a MOVSB instruction.) A byte is read from the address accessed by channel 0 and saved within the 8237 in a temporary holding register. Next, the 8237 initiates a memory write cycle where the contents of the temporary holding register are written into the address selected by DMA channel 1. The number of bytes transferred is determined by the channel 1 count register.

The channel 0 address hold enable bit (bit position 1) programs channel 0 for memory-to-memory transfers. For example, suppose you must fill an area of memory with data. Channel 0 can be held at the same address, while channel 1 changes for memory-to-memory transfer. This copies the contents of the address accessed by channel 0 into a block of memory accessed by channel 1.

The controller enable/disable bit (bit position 2) turns the entire controller on and off. The normal and compressed bit (bit position 3) determines whether a DMA cycle contains 2 (compressed) or 4 (normal) clocking periods. Bit position 5 is used

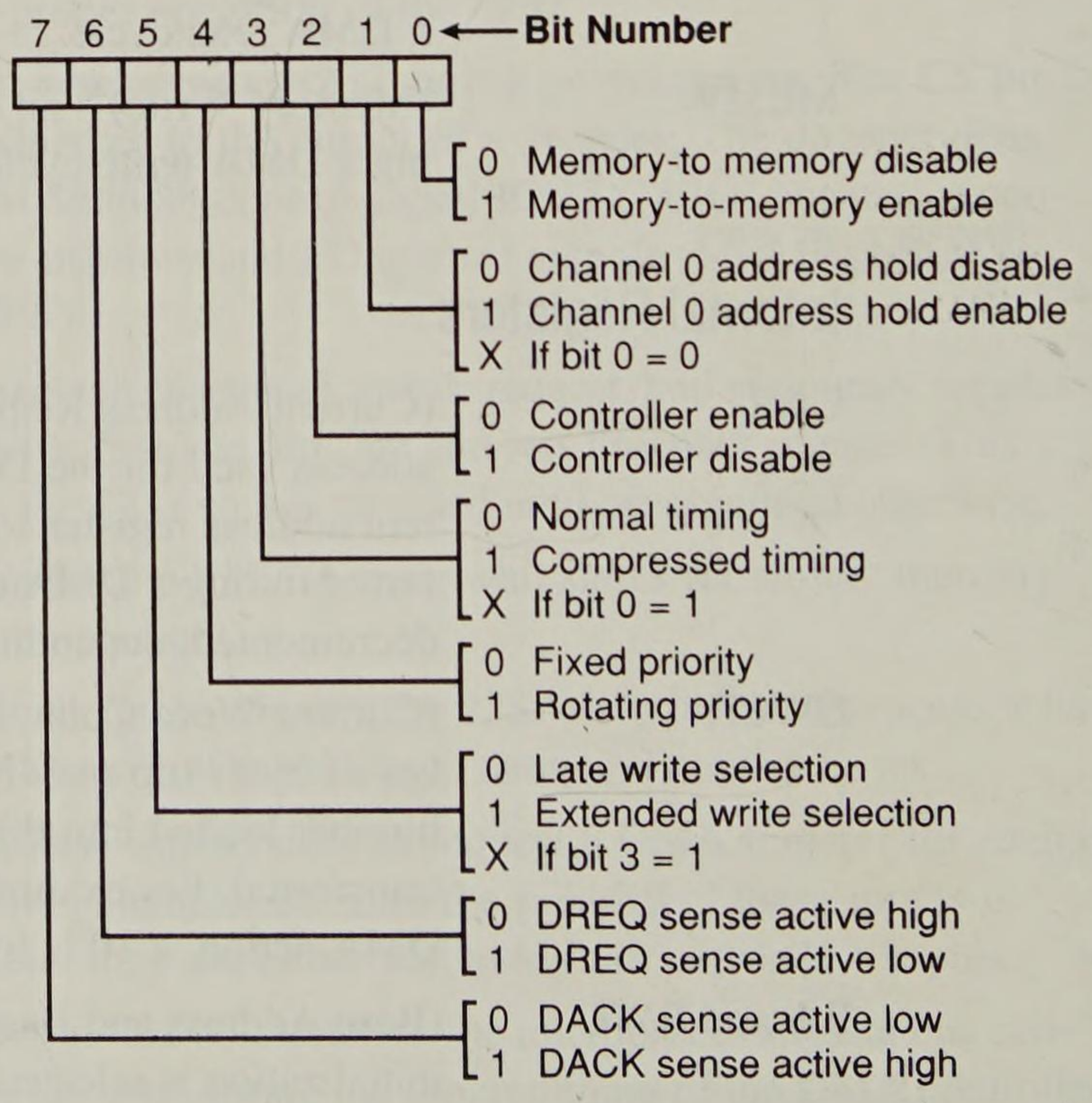



**FIGURE 13–4** 8237A-5 command register. (Courtesy of Intel Corporation)

in normal timing to extend the write pulse, so it appears one clock earlier in the timing for I/I devices that require a wider write pulse.

Bit position 4 selects priority for the 4 DMA channel DREQ inputs. In the fixed priority scheme, channel 0 has the highest priority and channel 3 has the lowest. In the rotating priority scheme, the most recently serviced channel assumes the lowest priority. For example, if channel 2 just had access to a DMA transfer, it assumes the lowest priority, and channel 3 assumes the highest priority position. Rotating priority is an attempt to give all channels equal priority.

The remaining two bits (bit positions 6 and 7) program the polarities of the DREQ inputs and the DACK outputs.

MR (Mode Register)—programs the mode of operation for a channel. Note that each channel has its own mode register as selected by bit positions 1 and 0 (see Figure 13–5). The remaining bits of the mode register select the operation, auto-initialization, increment/decrement, and mode for the channel. Verification operations generate the DMA addresses without generating the DMA memory and I/O control signals.

The modes of operation include demand mode, single mode, block mode, and cascade mode. Demand mode transfers data until an external $\overline{\text{EOP}}$ is input, or until the DREQ input becomes inactive. Single mode releases the HOLD after each byte of data is transferred. If the DREQ pin is held active, the 8237 again requests a DMA transfer through the DRQ line to the microprocessor's HOLD input. Block mode automatically transfers the number of bytes indicated by the count register for the channel. DREQ need not be held active through the block mode transfer. Cascade mode is used when more than one 8237 is present in a system.

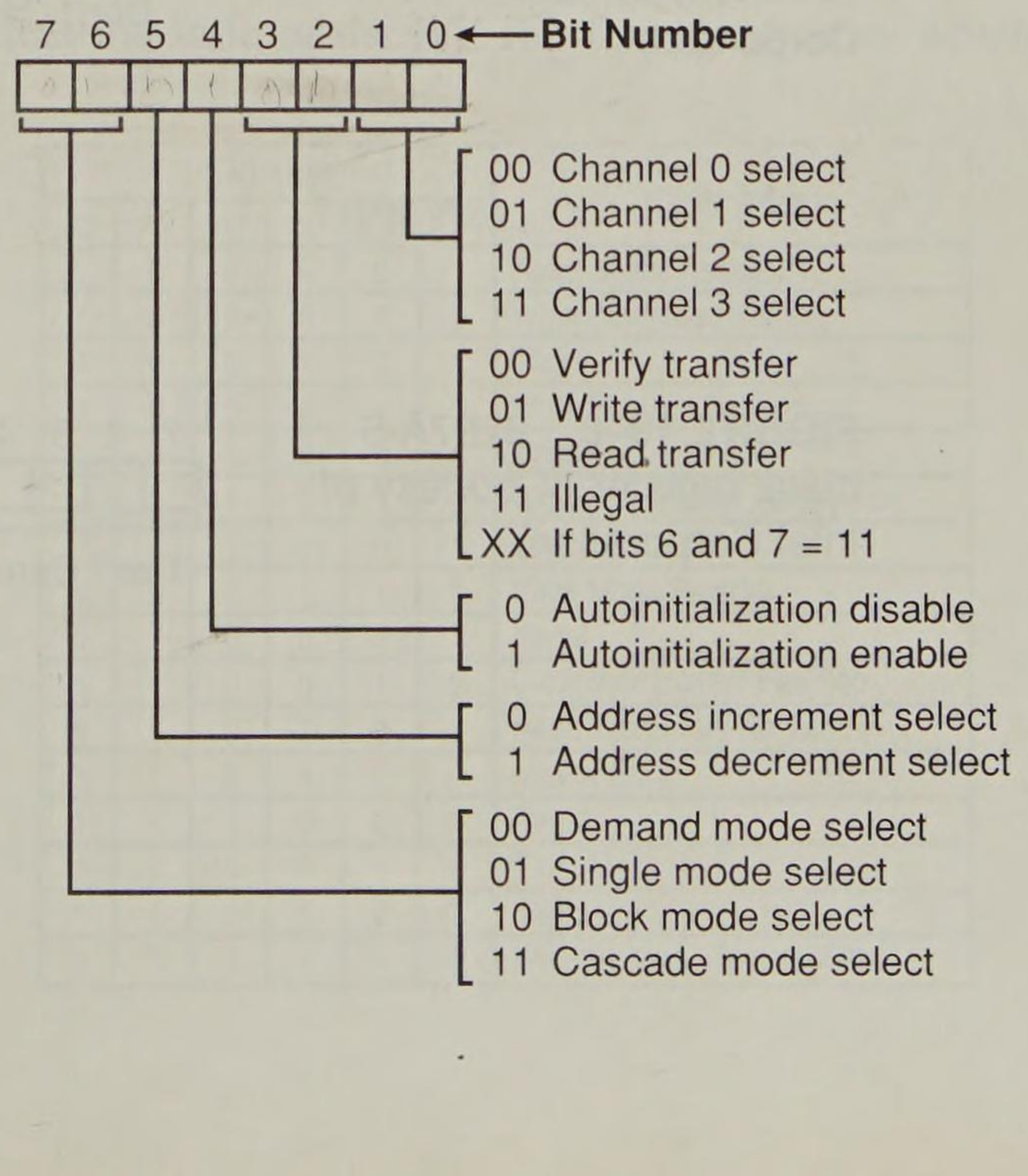



**FIGURE 13–5** 8237A-5 mode register. (Courtesy of Intel Corporation)

RR (Request Register)—is used to request a DMA transfer via software (see Figure 13–6). This is very useful in memory-to-memory transfers, where an external signal is not available to begin the DMA transfer.

MRSR (Mask Register Set/Reset)—sets or clears the channel mask as illustrated in Figure 13–7. If the mask is set, the channel is disabled. Recall that the RESET signal sets all channel masks to disable them.

MSR (Mask Register)—clears or sets all of the masks with one command, rather than individual channels like the MRSR. See Figure 13–8.

SR (Status Register)—shows the status of each DMA channel (see Figure 13–9). The TC bits indicate whether the channel has reached its terminal count (transferred all its bytes). Whenever the terminal count is reached, the DMA transfer is terminated for most modes of operation. The request bits indicate if the DREQ input for a given channel is active.

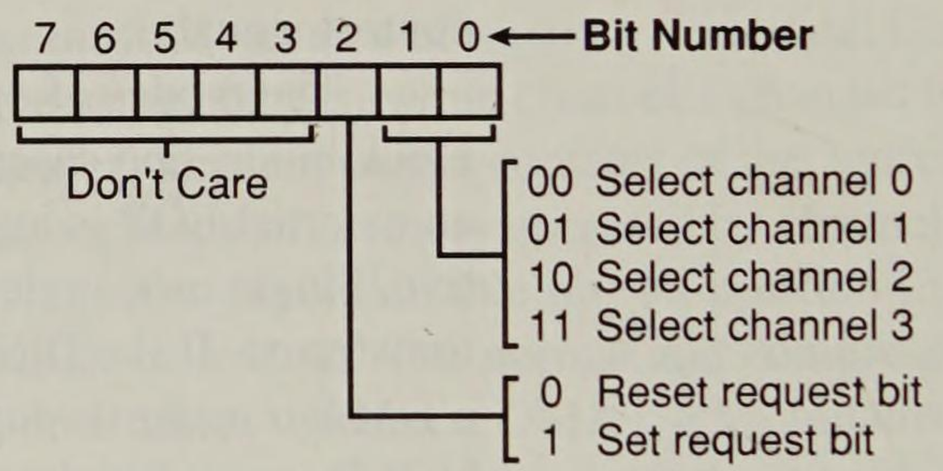



**FIGURE 13–6** 8237A-5 request register. (Courtesy of Intel Corporation)

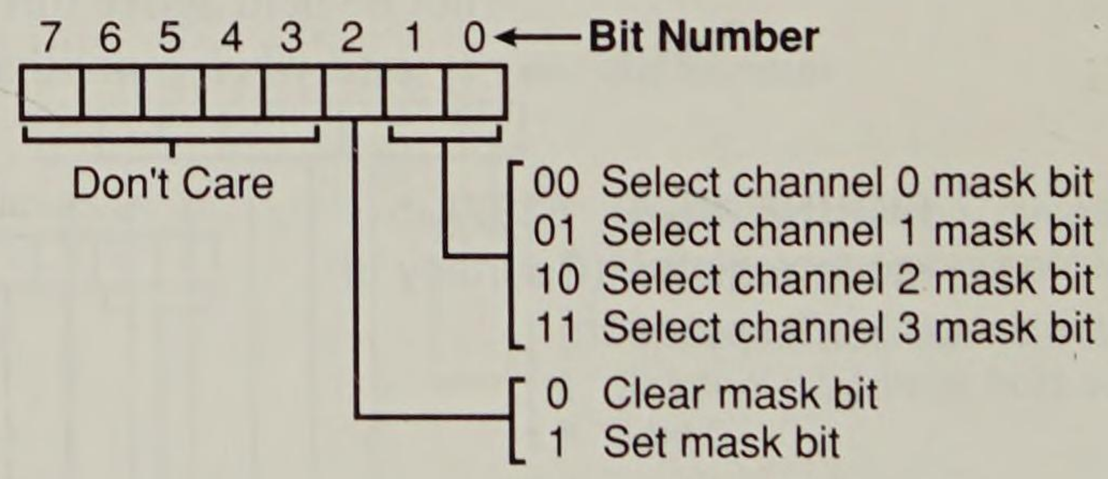



**FIGURE 13–7** 8237A-5 mask register set/reset mode. (Courtesy of Intel Corporation)

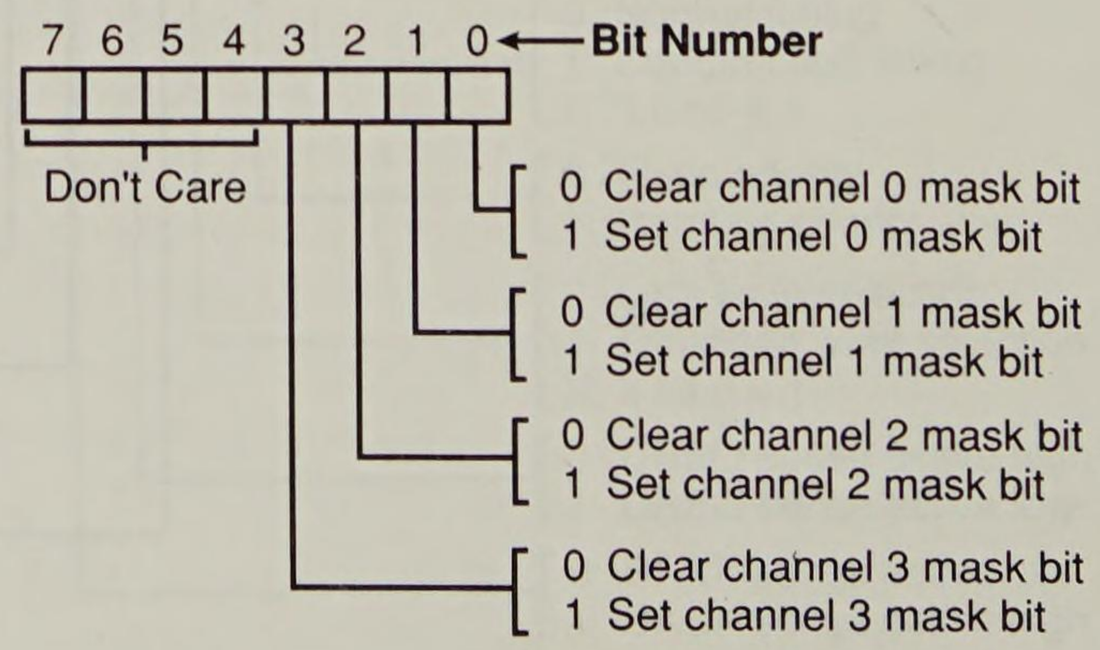



**FIGURE 13–8** 8237A-5 mask register. (Courtesy of Intel Corporation)

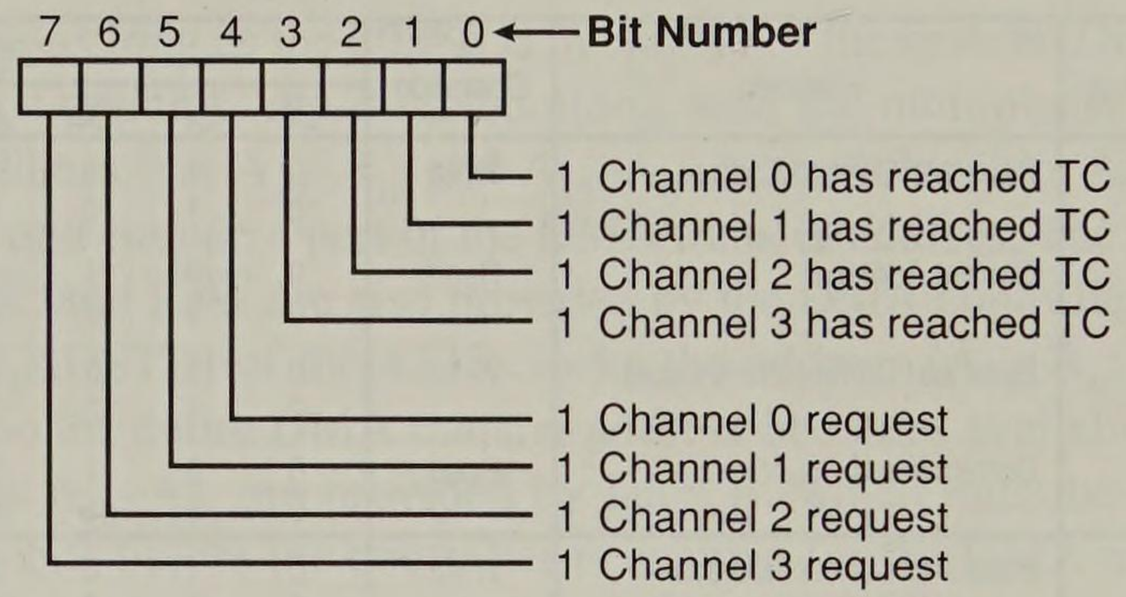



**FIGURE 13–9** 8237A-5 status register (Courtesy of Intel Corporation)

## Software Commands

Three software commands are used to control the operation of the 8237. These commands do not have a binary bit pattern as do the various control registers within the 8237. A simple output to the correct port number enables the software commands. Figure 13–10 shows the I/O port assignments that access all registers and the software commands.

The functions of the software commands are explained in the following list:

1. *Clear the first/last flip-flop*—clears the first/last (F/L) flip-flop within the 8237. The F/L flip-flop selects which byte (low or high order) is read/written in the current address and current count registers. If F/L = 0, the low-order byte is selected; if F/L = 1, the high-order byte is selected. Any read or write to the address or count register automatically toggles the F/L flip-flop.
2. *Master clear*—acts the same way as the RESET signal in the 8237. Like the RESET signal, this command disables all channels.
3. *Clear mask register*—enables all four DMA channels.

## Programming the Address and Count Registers

Figure 13–11 illustrates the I/O port locations for programming the count and address registers for each channel. Notice that the state of the F/L flip-flop determines whether the

**FIGURE 13–10** 8237A-5 command and control port assignments. (Courtesy of Intel Corporation)

| Signals | | | | | | Operation |
|---|---|---|---|---|---|---|
| A3 | A2 | A1 | A0 | $\overline{IOR}$ | $\overline{IOW}$ | |
| 1 | 0 | 0 | 0 | 0 | 1 | Read Status Register |
| 1 | 0 | 0 | 0 | 1 | 0 | Write Command Register |
| 1 | 0 | 0 | 1 | 0 | 1 | Illegal |
| 1 | 0 | 0 | 1 | 1 | 0 | Write Request Register |
| 1 | 0 | 1 | 0 | 0 | 1 | Illegal |
| 1 | 0 | 1 | 0 | 1 | 0 | Write Single Mask Register Bit |
| 1 | 0 | 1 | 1 | 0 | 1 | Illegal |
| 1 | 0 | 1 | 1 | 1 | 0 | Write Mode Register |
| 1 | 1 | 0 | 0 | 0 | 1 | Illegal |
| 1 | 1 | 0 | 0 | 1 | 0 | Clear Byte Pointer Flip/Flop |
| 1 | 1 | 0 | 1 | 0 | 1 | Read Temporary Register |
| 1 | 1 | 0 | 1 | 1 | 0 | Master Clear |
| 1 | 1 | 1 | 0 | 0 | 1 | Illegal |
| 1 | 1 | 1 | 0 | 1 | 0 | Clear Mask Register |
| 1 | 1 | 1 | 1 | 0 | 1 | Illegal |
| 1 | 1 | 1 | 1 | 1 | 0 | Write All Mask Register Bits |

| Channel | Register | Operation | Signals | | | | | | | Internal Flip-Flop | Data Bus DB0-DB7 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\overline{CS}$ | $\overline{IOR}$ | $\overline{IOW}$ | A3 | A2 | A1 | A0 | | |
| 0 | Base and Current Address | Write | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | A0-A7 |
| | | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | A8-A15 |
| | Current Address | Read | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | A0-A7 |
| | | | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | A8-A15 |
| | Base and Current Word Count | Write | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | W0-W7 |
| | | | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | W8-W15 |
| | Current Word Count | Read | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | W0-W7 |
| | | | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | W8-W15 |
| 1 | Base and Current Address | Write | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | A0-A7 |
| | | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | A8-A15 |
| | Current Address | Read | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | A0-A7 |
| | | | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | A8-A15 |
| | Base and Current Word Count | Write | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | W0-W7 |
| | | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | W8-W15 |
| | Current Word Count | Read | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | W0-W7 |
| | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | W8-W15 |
| 2 | Base and Current Address | Write | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | A0-A7 |
| | | | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | A8-A15 |
| | Current Address | Read | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | A0-A7 |
| | | | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | A8-A15 |
| | Base and Current Word Count | Write | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | W0-W7 |
| | | | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | W8-W15 |
| | Current Word Count | Read | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | W0-W7 |
| | | | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | W8-W15 |
| 3 | Base and Current Address | Write | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | A0-A7 |
| | | | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | A8-A15 |
| | Current Address | Read | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | A0-A7 |
| | | | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | A8-A15 |
| | Base and Current Word Count | Write | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | W0-W7 |
| | | | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | W8-W15 |
| | Current Word Count | Read | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | W0-W7 |
| | | | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | W8-W15 |

**FIGURE 13–11** 8237A-5 DMA channel I/O port addresses. (Courtesy of Intel Corporation)

LSB or MSB is programmed. If the state of the F/L flip-flop is unknown, the count and address could be programmed incorrectly. It is also important that the DMA channel is disabled before its address and count are programmed.

There are four steps required to program the 8237: (1) the F/L flip-flop is cleared using a clear F/L command, (2) the channel is disabled, (3) the LSB and then MSB of the address are programmed, and (4) the LSB and MSB of the count are programmed. Once these four operations are performed, the channel is programmed and ready to use. Additional programming is required to select the mode of operation before the channel is enabled and started.

## The 8237 Connected to the 80×86 Microprocessor

Figure 13–12 shows an 80×86-based system that contains the 8237 DMA controller.

The address enable (AEN) output of the 8237 controls the output pins of the latches and the outputs of the 74LS257 (E). During normal 80×86 operation (AEN = 0), latches A and C and the multiplexer (E) provide address bus bits $A_{19}$–$A_{16}$ and $A_7$–$A_0$. The multiplexer

provides the system control signals as long as the 80×86 is in control of the system. During a DMA action (AEN = 1), latches A and C are disabled along with the multiplexer (E). Latches D and B now provide address bits $A_{19}$–$A_{16}$ and $A_{15}$–$A_8$. Address bus bits $A_7$–$A_0$ are provided directly by the 8237 and contain a part of the DMA transfer address. The control signals $\overline{\text{MEMR}}$, $\overline{\text{MEMW}}$, $\overline{\text{IOR}}$, and $\overline{\text{IOW}}$ are also provided by the DMA controller.

The address strobe output (ADSTB) of the 8237 clocks the address ($A_{15}$–$A_8$) into latch D during the DMA action, so the entire DMA transfer address becomes available on the address bus. Address bus bits $A_{19}$–$A_{16}$ are provided by latch B, which must be programmed with these four address bits before the controller is enabled for the DMA transfer. The DMA operation of the 8237 is limited to a transfer of not more than 64K bytes within the same 64K-byte section of the memory.

The decoder (F) selects the 8237 for programming and also the 4-bit latch (B) for the upper-most four address bits. The decoder in this system enables the 8237 for I/O port addresses XX60H–XX6FH and the I/O latch (B) for ports XX00H–XX0FH. Notice that the decoder output is combined with the $\overline{\text{IOW}}$ signal to generate an active-high clock for the latch (B).

During normal 80×86 operation, the DMA controller and integrated circuits B and D are disabled. During a DMA action, integrated circuits A, C, and E are disabled so the 8237 can take control of the system through the address, data, and control buses.

In the personal computer, the two DMA controllers are programmed at I/O ports 0000H–000FH for DMA channels 0–3 and I/O ports 00C0H–00DFH for DMA channels 4–7. Note that the second controller is programmed at even addresses only, so channel 4 base and current address is programmed at I/O port 00C0H and the channel 4 base and current count is programmed at port 00C2H. The page register, which holds address bits A23–A16 of the DMA address, is located at I/O ports 0087H (CH–0), 0083H (CH–1), 0081H (CH–2), 0082H (CH–3), (no channel 4), 008BH (CH–5), 0089H (CH–6) and 008AH (CH–7). The page register functions as the address latch described with the examples in this text.

## Memory-to-Memory Transfer with the 8237

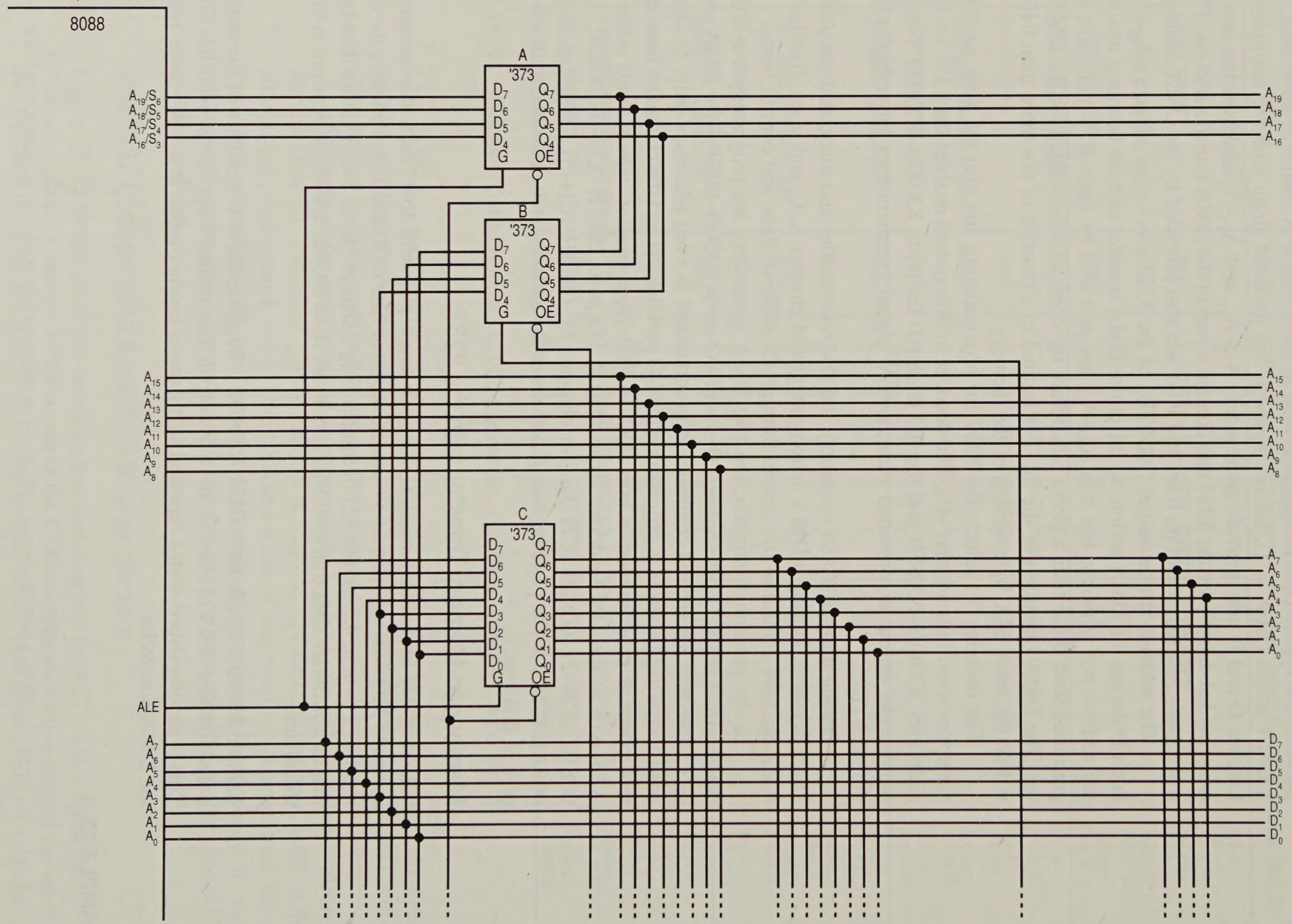
The memory-to-memory transfer is much more powerful than even the automatically repeated MOVSB instruction. While the repeated MOVSB instruction enables the 8088 to transfer a byte in 4.2 µs, the 8237 requires only 2.0 µs per byte. This is more than twice as fast as a software data transfer. This is not true if an 80386, 80846, or Pentium is in use in the system.

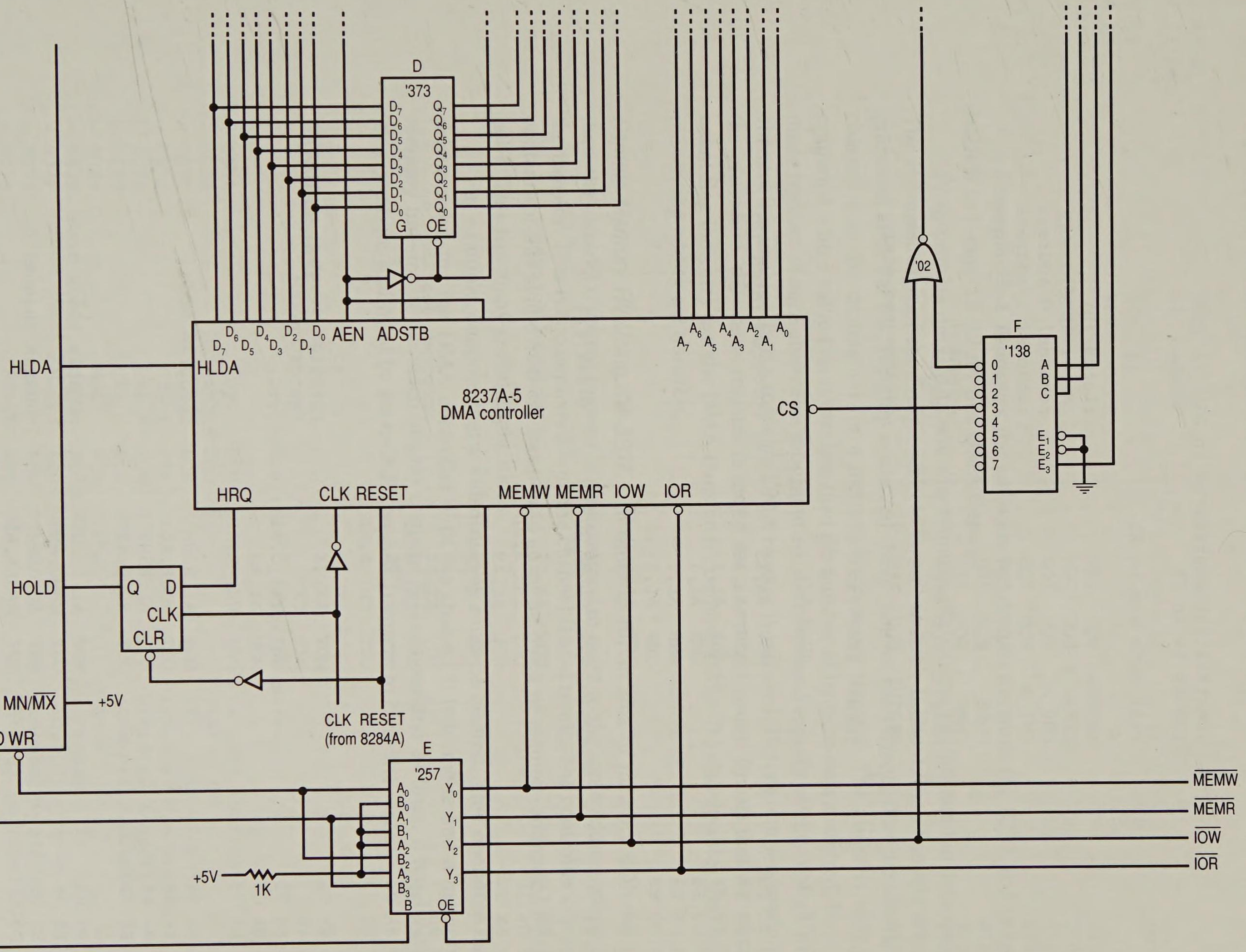
*Sample Memory-to-Memory DMA Transfer.* Suppose that the contents of memory locations 10000H–13FFFH are to be transferred into memory locations 14000H–17FFFH. This is accomplished with a repeated string move instruction or, at a much faster rate, with the DMA controller.

**EXAMPLE 13–1**

```
;Procedure that transfers a block of data using the 8237A
;DMA controller.  This is a memory-to-memory block transfer.
;
;Source address is in SI
```

**FIGURE 13–12** Complete 8088 minimum mode DMA system. (pp. 570–571)




D
'373
$D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$
G OE
$Q_7$ $Q_6$ $Q_5$ $Q_4$ $Q_3$ $Q_2$ $Q_1$ $Q_0$
'02
F
'138
0 1 2 3 4 5 6 7
A B C
$E_1$ $E_2$ $E_3$
HLDA
$D_7$ $D_6$ $D_5$ $D_4$ $D_3$ $D_2$ $D_1$ $D_0$ AEN ADSTB
$A_7$ $A_6$ $A_5$ $A_4$ $A_3$ $A_2$ $A_1$ $A_0$
HLDA
8237A-5
DMA controller
CS
HRQ CLK RESET MEMW MEMR IOW IOR
HOLD
Q D
CLK
CLR
MN/$\overline{MX}$ +5V
IO/$\overline{M}$ RD WR
CLK RESET
(from 8284A)
E
'257
$A_0$ $B_0$ $A_1$ $B_1$ $A_2$ $B_2$ $A_3$ $B_3$
$Y_0$ $Y_1$ $Y_2$ $Y_3$
B OE
+5V
1K
$\overline{MEMW}$
$\overline{MEMR}$
$\overline{IOW}$
$\overline{IOR}$

```
                        ;Destination address is in DI
                        ;Count is in CX
                        ;
                        ;All data are in ES
                        ;
= 0010                  LATCHB  EQU  10H             ;Latch (B)
= 007C                  CLEAR_F EQU  7CH             ;F/L flip-flop
= 0070                  CH0_A   EQU  70H             ;channel 0 address
= 0072                  CH1_A   EQU  72H             ;channel 1 address
= 0073                  CH1_C   EQU  73H             ;channel 1 count
= 007B                  MODE    EQU  7BH             ;mode
= 0078                  CMMD    EQU  78H             ;command
= 007F                  MASKS   EQU  7FH             ;mask
= 0079                  REQ     EQU  79H             ;request register
= 0078                  STATUS  EQU  78H             ;status register

0000                    TRANS   PROC FAR

0000 50                         PUSH AX

0001 8C C0                      MOV  AX,ES           ;program latch (B)
0003 8A C4                      MOV  AL,AH
0005 D0 E8                      SHR  AL,1
0007 D0 E8                      SHR  AL,1
0009 D0 E8                      SHR  AL,1
000B D0 E8                      SHR  AL,1
000D E6 10                      OUT  LATCHB,AL

000F E6 7C                      OUT  CLEAR_F,AL      ;clear F/L

0011 8B C6                      MOV  AX,SI           ;program source
0013 E6 70                      OUT  CH0_A,AL
0015 8A C4                      MOV  AL,AH
0017 E6 70                      OUT  CH0_A,AL

0019 8B C7                      MOV  AX,DI           ;program destination
001B E6 72                      OUT  CH1_A,AL
001D 8A C4                      MOV  AL,AH
001F E6 72                      OUT  CH1_A,AL

0021 8B C1                      MOV  AX,CX           ;program count
0023 48                         DEC  CX              ;adjust count
0024 E6 73                      OUT  CH1_C,AL
0026 8A C4                      MOV  AL,AH
0028 E6 73                      OUT  CH1_C,AL

002A B0 88                      MOV  AL,88H          ;program mode
002C E6 7B                      OUT  MODE,AL
002E B0 85                      MOV  AL,85H
0030 E6 7B                      OUT  MODE,AL

0032 B0 01                      MOV  AL,1            ;enable block move
0034 E6 78                      OUT  CMMD,AL
0036 B0 0E                      MOV  AL,0EH          ;unmask channel 0
0038 E6 7F                      OUT  MASKS,AL

003A B0 04                      MOV  AL,4            ;request DMA
003C E6 79                      OUT  REQ,AL

003E                    AGAIN:

003E E4 78                      IN   AL,STATUS       ;wait until DMA complete
```

```
0040 A8 01                      TEST AL,1
0042 74 FA                      JZ   AGAIN

0044 58                         POP  AX
0046 CB                         RET

0046                    TRANS   ENDP
```

Example 13–1 illustrates the software required to initialize the 8237 and program latch B in Figure 13–12 for this DMA transfer.

Programming the DMA controller requires a few steps, as illustrated in the example. The left-most digit of the 5-digit address is sent to latch B. Next, the channels are programmed after the F/L flip-flop is cleared. Note that we use channel 0 as the source and channel 1 as the destination for a memory-to-memory transfer. The count is next programmed with a value that is one less than the number of bytes to be transferred. Next, the mode register of each channel is programmed, the command register selects a block move, channel 0 is enabled, and a software DMA request is initiated. Before return is made from the procedure, the status register is tested for a terminal count. Recall that the terminal count flag indicates that the DMA transfer is completed. The TC also disables the channel preventing additional transfers.

*Sample Memory Fill Using the 8237.* In order to fill an area of memory with the same data, the channel 0 source register is programmed to point to the same address throughout the transfer. This is accomplished with the channel 0 hold mode. The controller copies the contents of this single memory location to an entire block of memory addressed by channel 1. This has many useful applications.

For example, suppose that a video display must be cleared. This operation can be performed using the DMA controller with the channel 0 hold mode and a memory-to-memory transfer. If the video display contains 80 columns and 24 lines, it has 1,920 display positions that must be set to 20H (an ASCII space) to clear the screen.

**EXAMPLE 13–2**

```
                        ;Procedure that clears the screen
                        ;
= 0002                  ADDR    EQU  2              ;screen address
= 0010                  LATCHB  EQU  10H            ;latch (B)
= 007C                  CLR_F   EQU  7CH            ;F/L flip-flop
= 0070                  CH0_A   EQU  70H            ;channel 0 address
= 0072                  CH1_A   EQU  72H            ;channel 1 address
= 0073                  CH1_C   EQU  73H            ;channel 1 count
= 007B                  MODE    EQU  7BH            ;mode
= 0078                  CMMD    EQU  78H            ;command
= 007F                  MASKS   EQU  7FH            ;mask
= 0079                  REQ     EQU  79H            ;request register
= 0078                  STATUS  EQU  78H            ;status register
= 0020                  SPACE   EQU  20H            ;ASCII space

0000                    CLEAR   PROC FAR

0000 50                         PUSH AX
0001 06                         PUSH ES
0002 53                         PUSH BX
```

```
0003 B0 02                      MOV   AL,ADDR
0005 E6 10                      OUT   LATCHB,AL
0007 D0 E0                      SHR   AL,1
0009 D0 E0                      SHR   AL,1
000B D0 E0                      SHR   AL,1
000D D0 E0                      SHR   AL,1
000F 8A E0                      MOV   AH,AL
0011 32 C0                      XOR   AL,AL
0013 8E C0                      MOV   ES,AX             ;address segment

0015 E6 7C                      OUT   CLR_F,AL          ;clear F/L

0017 B8 0014 R                  MOV   AX,OFFSET SCREEN
001A 8B D8                      MOV   BX,AX
001C 26: C6 07 20               MOV   BYTE PTR ES:[BX],SPACE

0020 E6 70                      OUT   CH0_A,AL
0022 8A C4                      MOV   AL,AH
0024 E6 70                      OUT   CH0_A,AL

0026 8B C3                      MOV   AX,BX
0028 40                         INC   AX
0029 E6 72                      OUT   CH1_A,AL
002B 8A C4                      MOV   AL,AH
002D E6 72                      OUT   CH1_A,AL
002F B8 077E                    MOV   AX,1918           ;program count
0032 E6 73                      OUT   CH1_C,AL
0034 8A C4                      MOV   AL,AH
0036 E6 73                      OUT   CH1_C,AL

0038 B0 88                      MOV   AL,88H            ;program mode
003A E6 7B                      OUT   MODE,AL
003C B0 85                      MOV   AL,85H
003E E6 7B                      OUT   MODE,AL
0040 B0 03                      MOV   AL,3              ;program copy
0042 E6 78                      OUT   CMMD,AL

0044 B0 0E                      MOV   AL,0EH            ;unmask channel 0
0046 E6 7F                      OUT   MASKS,AL

0048 B0 04                      MOV   AL,4              ;request DMA
004A E6 79                      OUT   REQ,AL

004C                    AGAIN:

004C E4 78                      IN    AL,STATUS         ;wait until DMA complete
004E A8 01                      TEST  AL,1
0050 74 FA                      JZ    AGAIN

0052 6B                         POP   BX
0053 07                         POP   ES
0054 58                         POP   AX
0055 CB                         RET

0056                    CLEAR   ENDP
```

Example 13–2 shows a procedure that clears the video display. Here the screen address is known, as is the number of bytes on the screen. Notice that this procedure is nearly identical to Example 13–1, except the command register is programmed so the channel 0 address is held.

## DMA-Processed Printer Interface

Figure 13–13 illustrates the hardware added to Figure 13–12 for a DMA-controlled printer interface. Little additional circuitry is added for this interface to a Centronics-type parallel printer. The latch is used to capture the data as it is sent to the printer during the DMA transfer. The write pulse passed through to the latch during the DMA action also generates the data strobe ($\overline{DS}$) signal to the printer through the single-shot. The $\overline{ACK}$ signal returns from the printer each time it is ready for additional data. In this circuit, $\overline{ACK}$ is used to request a DMA action through a flip-flop.

Notice that the I/O device is not selected by decoding the address on the address bus. During the DMA transfer, the address bus contains the memory address and cannot contain the I/O port address. In place of the I/O port address, the $\overline{DACK}_3$ output from the 8237 selects the latch by gating the write pulse through an OR gate.

Software that controls this interface is simple, because only the address of the data and the number of characters to be printed are programmed. Once programmed, the channel is enabled, and the DMA action transfers a byte at a time to the printer interface each time the interface receives the $\overline{ACK}$ single from the printer.

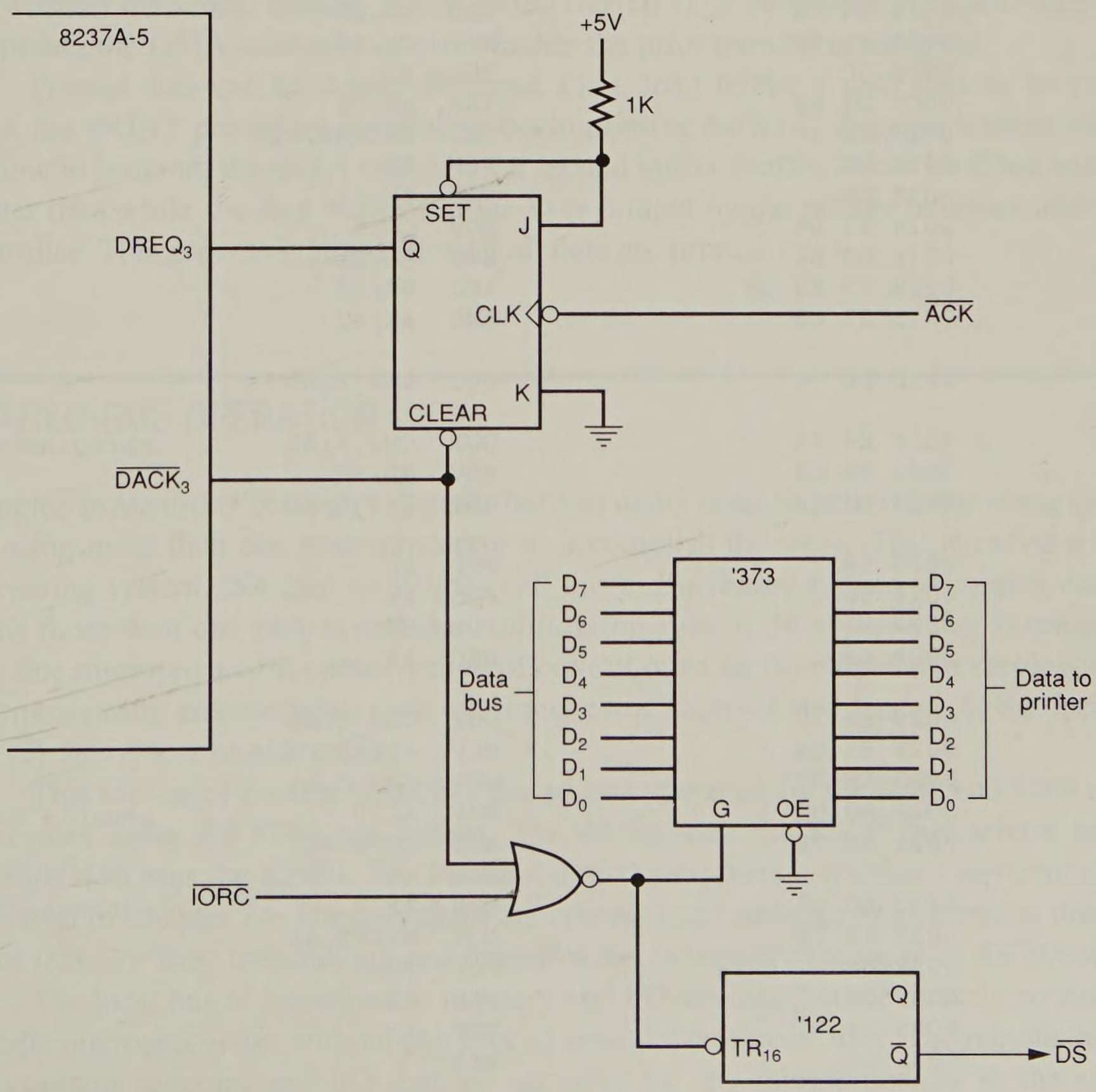



**FIGURE 13–13** DMA process printer interface.

**EXAMPLE 13–3**

```
                        ;Procedure that prints data beginning at the location
                        ;addressed by DS:BX.  The number of characters printed
                        ;are in CX.
                        ;
= 0010                  LATCHB EQU  10H             ;latch (B)
= 007C                  CLR_F  EQU  7CH             ;F/L flip-flop
= 0076                  CH3_A  EQU  72H             ;channel 3 address
= 0077                  CH3_C  EQU  73H             ;channel 3 count
= 007B                  MODE   EQU  7BH             ;mode
= 0078                  CMMD   EQU  78H             ;command
= 007F                  MASKS  EQU  7FH             ;mask
= 0079                  REQ    EQU  79H             ;request register

0000                    PRINT  PROC FAR

0000 50                        PUSH AX
0001 51                        PUSH CX
0002 53                        PUSH BX

0003 8C D8                     MOV  AX,DS           ;program A19-A16
0005 B1 04                     MOV  CL,4
0007 D3 EB                     SHR  BX,CL
0009 03 C3                     ADD  AX,BX
000B B1 0C                     MOV  CL,12
000D 50                        PUSH AX
000E D3 E8                     SHR  AX,CL
0010 E6 10                     OUT  LATCHB,AL
0012 58                        POP  AX
0013 5B                        POP  BX
0014 B1 04                     MOV  CL,4
0016 D3 E0                     SHR  AX,CL
0018 83 E3 0F                  AND  BX,15
001B 03 C3                     AND  AX,BX

001D E6 7C                     OUT  CLR_F,AL

001F E6 76                     OUT  CH3_A,AL        ;program address
0021 8A C4                     MOV  AL,AH
0023 E6 76                     OUT  CH3_A,AL

0025 58                        POP  AX
0026 50                        PUSH AX
0027 53                        PUSH BX
0028 48                        DEC  AX

0029 E6 77                     OUT  CH3_C,AL
002B 8A C4                     MOV  AL,AH
002D E6 77                     OUT  CH3_C,AL
002F B0 00                     MOV  AL,0            ;enable controller
0031 E6 78                     OUT  CMMD,AL

0033 B0 07                     MOV  AL,7            ;unmask channel 3
0035 E6 7F                     OUT  MASKS,AL
0037 5B                        POP  BX
0038 59                        POP  CX
0039 58                        POP  AX
003A CB                        RET

003B                    PRINT  ENDP
```

The procedure that prints data from the current data segment is illustrated in Example 13–3. This procedure programs the 8237, but doesn't actually print anything. Printing is accomplished by the DMA controller and the printer interface.

**EXAMPLE 13–4**

```
                        ;Procedure to test for a completed DMA action.
                        ;
= 0078                  STATUS EQU  78H              ;status register

0000                    TEST_P PROC NEAR

0000 E4 78                     IN   AL,STATUS        ;test channel 3
0002 A8 08                     TEST AL,8
0004 74 FA                     JZ   TEST_P
0006 C3                        RET

0007                    TEST_P ENDP
```

A secondary procedure is needed to determine whether the DMA action has been completed. Example 13–4 lists the secondary procedure that tests the DMA controller to see whether the DMA transfer is complete. The TEST_P procedure is called before programming the DMA controller to see whether the prior transfer is complete.

Printed data can be double-buffered. First, load buffer 1 with data to be printed. Next, the PRINT procedure is called to begin printing buffer 1. Because it takes very little time to program the DMA controller, a second buffer (buffer 2) can be filled with new printer data while the first buffer (buffer 1) is printed by the printer interface and DMA controller. This process is repeated until all data are printed.

## 13–3 SHARED-BUS OPERATION

Complex present-day computer systems have so many tasks to perform that some systems are using more than one microprocessor to accomplish the work. This is called a *multiprocessing* system. We also sometimes call this a *distributed* system. A system that performs more than one task is called a *multitasking* system. In systems that contain more than one microprocessor, some method of control must be developed and employed. In a multiprocessing environment, each microprocessor accesses two buses: (1) the *local bus* and (2) the *remote* or *shared bus*.

This section of the text describes shared-bus operation for the 8086 and 8088 microprocessors using the 8289 bus arbiter. The 80286 uses the 82289 bus arbiter and the 80386/80486 uses the 82389. The Pentium directly supports a multiuser environment, as described in Chapter 14. These systems are complex and difficult to illustrate at this point in the text, but their terminology and operation are essentially the same as the 8086/8088.

The local bus is connected to memory and I/O devices that are directly accessed by a single microprocessor, without any special protocol or access rules. The remote (shared) bus contains memory and I/O that are accessed by any microprocessor in the system. Figure 13–14 illustrates this idea with a few microprocessors.

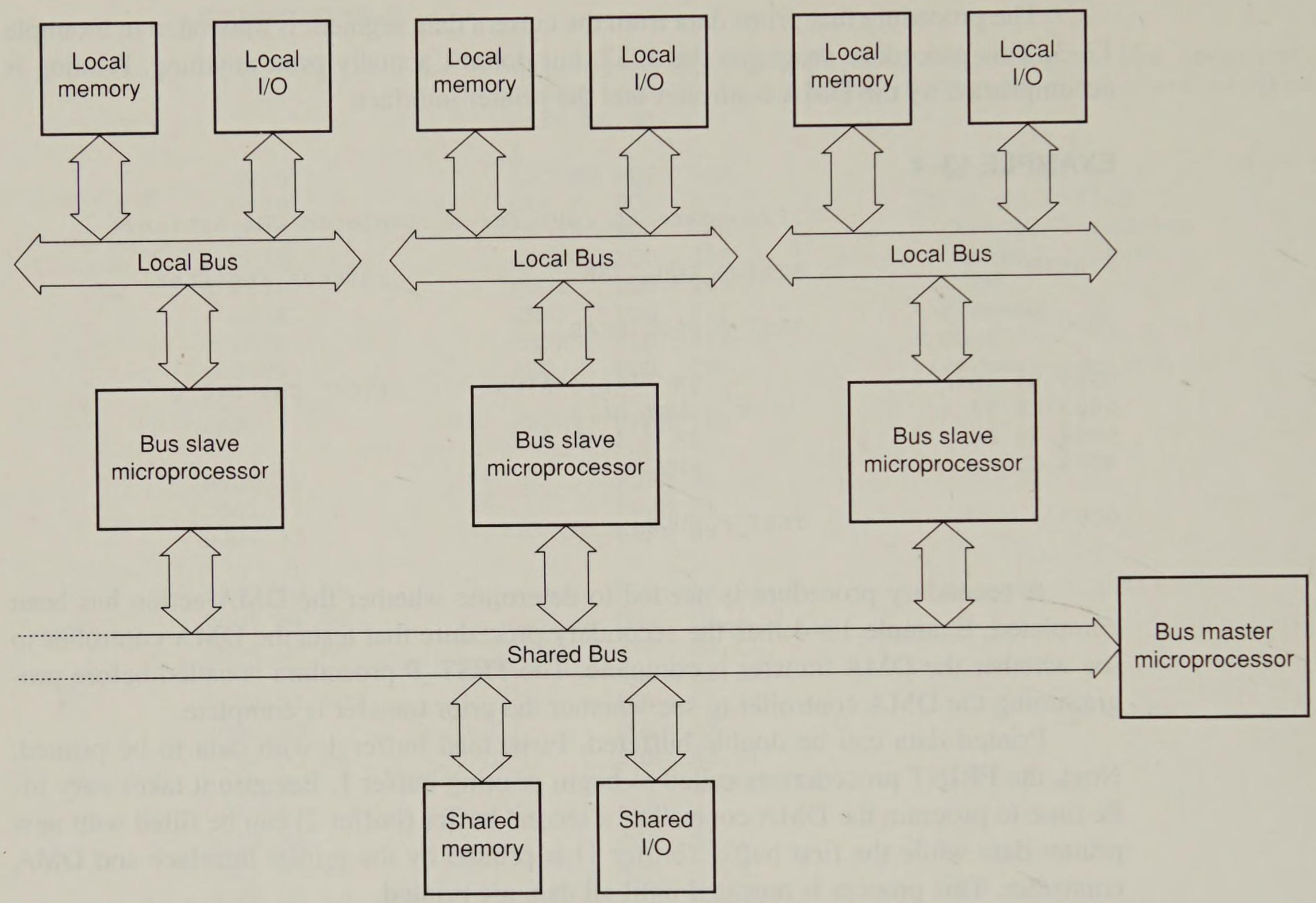



**FIGURE 13–14** A block diagram illustrating the shared and local buses.

## Types of Buses

The local bus is the bus that is resident to the microprocessor. The local bus contains the resident or local memory and I/O. All microprocessors studied thus far in this text are considered local bus systems. The local memory and local I/O are accessed by the microprocessor that is directly connected to them.

A shared bus is one that is connected to all microprocessors in the system. The shared bus is used to exchange data between microprocessors in the system. A shared bus may contain memory and I/O devices that are accessed by all microprocessors in the system. Access to the shared bus is controlled by some form of arbiter that allows only a single microprocessor to access the system's shared bus space.

Figure 13–15 shows an 8088 microprocessor connected as a remote bus master. The term *bus master* applies to any device (microprocessor or otherwise) that can control a bus containing memory and I/O. The 8237 DMA controller presented earlier in the chapter is an example of a remote bus master. The DMA controller gained access to the system memory and I/O space to cause a data transfer. Likewise, a remote bus master gains access to the shared bus for the same purpose. The difference is that the remote bus master microprocessor can execute variable software, whereas the DMA controller can only transfer data.

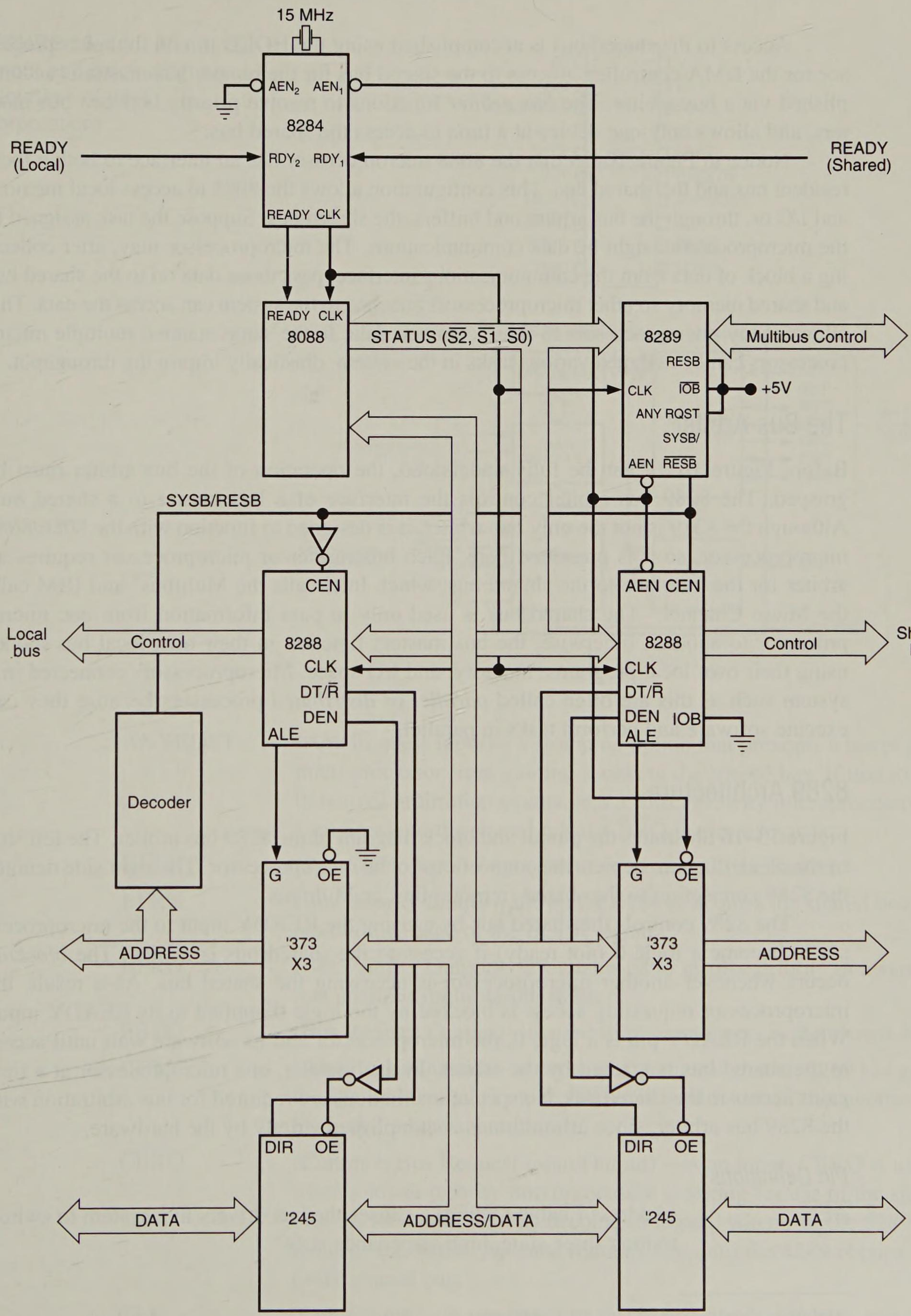



**FIGURE 13–15** The 8088 operated in the remote mode illustrating the local and shared bus connections.

Access to the shared bus is accomplished using the HOLD pin on the microprocessor for the DMA controller. Access to the shared bus for the remote bus master is accomplished via a bus arbiter. The *bus arbiter* functions to resolve priority between bus masters, and allows only one device at a time to access the shared bus.

Notice in Figure 10–15 that the 8088 microprocessor has an interface to both a local resident bus and the shared bus. This configuration allows the 8088 to access local memory and I/O or, through the bus arbiter and buffers, the shared bus. Suppose the task assigned to the microprocessor might be data communications. The microprocessor may, after collecting a block of data from the communications interface, pass those data on to the shared bus and shared memory so other microprocessors attached to the system can access the data. This allows many microprocessors to share common data. In the same manner, multiple microprocessors can be assigned various tasks in the system, drastically improving throughput.

## The Bus Arbiter

Before Figure 10–17 can be fully understood, the operation of the bus arbiter must be grasped. The 8289 bus arbiter controls the interface of a bus master to a shared bus. Although the 8289 is not the only bus arbiter, it is designed to function with the 8086/8088 microprocessor, so it is presented here. Each bus master or microprocessor requires an arbiter for the interface to the shared bus, which Intel calls the Multibus* and IBM calls the Micro Channel.† The shared bus is used only to pass information from one microprocessor to another; otherwise, the bus masters function in their own local bus modes using their own local programs, memory, and I/O space. Microprocessors connected in a system such as this are often called *parallel* or *distributed* processors because they can execute software and perform tasks in parallel.

## 8289 Architecture

Figure 13–16 illustrates the pinout and block diagram of the 8289 bus arbiter. The left side of the block diagram depicts the connections to the microprocessor. The right side denotes the 8289 connection to the shared (remote) bus, or Multibus.

The 8289 controls the shared bus by causing the READY input to the microprocessor to become a logic 0 (not ready) if access to the shared bus is denied. The *blocking* occurs whenever another microprocessor is accessing the shared bus. As a result, the microprocessor requesting access is blocked by the logic 0 applied to its READY input. When the READY pin is a logic 0, the microprocessor and its software wait until access to the shared bus is granted by the arbiter. In this manner, one microprocessor at a time gains access to the shared bus. No special instructions are required for bus arbitration with the 8289 bus arbiter, since arbitration is accomplished strictly by the hardware.

### *Pin Definitions*

$\overline{\text{AEN}}$ (Address Enable Output)—causes the bus drivers in a system to switch to their three-state, high-impedance state.

---

*Multibus is a trademark of Intel Corporation.

†Micro Channel is a trademark of International Business Machines.

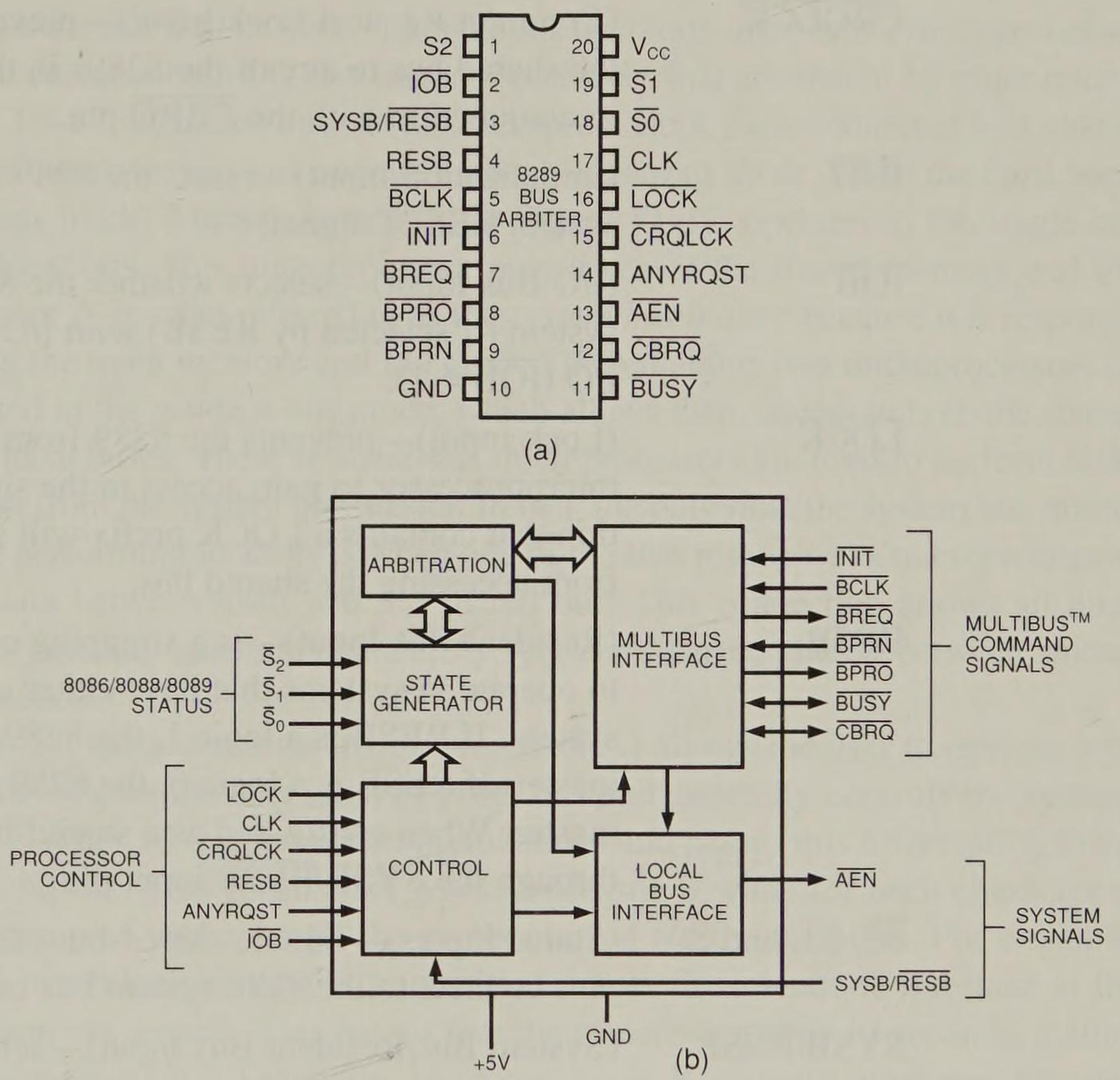



**FIGURE 13–16** The 8289 pinout and block diagram. (Courtesy of Intel Corporation)

ANYRQST (Any Request Input)—a strapping option that prevents a lower priority microprocessor from gaining access to the shared bus. If tied to a logic 0, normal arbitration occurs, and a lower priority microprocessor can gain access to the shared bus if $\overline{\text{CBRQ}}$ is also a logic 0.

$\overline{\text{BCLK}}$ (Bus Clock Input)—synchronizes all shared-bus masters.

$\overline{\text{BPRN}}$ (Bus Priority Input)—allows the 8289 to acquire the shared bus on the next falling edge of the $\overline{\text{BCLK}}$ signal.

$\overline{\text{BPRO}}$ (Bus Priority Output)—a signal used to resolve priority in a system that contains multiple-bus masters.

$\overline{\text{BREQ}}$ (Bus Request Output)—is used to request access to the shared bus.

$\overline{\text{BUSY}}$ (Busy Input/Output)—as an output, indicates that an 8289 has acquired the shared bus. As an input, $\overline{\text{BUSY}}$ is used to detect that another 8289 has acquired the shared bus.

$\overline{\text{CBRQ}}$ (Common Bus Request Input/Output)—as an input, CBRQ is used when a lower priority microprocessor is asking for use of the shared bus. As an output, $\overline{\text{CBRQ}}$ becomes a logic 0 whenever the 8289 requests the shared bus, and remains low until the 8289 obtains access to the shared bus.

CLK (Clock Input)—is generated by the 8284A clock generator and provides the internal timing source to the 8289.

$\overline{\text{CRQLCK}}$ (Common Request Lock Input)—prevents the 8289 from surrendering the shared bus to any of the 8289s in the system. This signal functions in conjunction with the $\overline{\text{CBRQ}}$ pin.

$\overline{\text{INIT}}$ (Initialization Input)—resets the 8289 and is normally connected to the system RESET signal.

$\overline{\text{IOB}}$ (I/O Bus Input)—selects whether the 8289 operates in a shared-bus system (if selected by RESB) with I/O ($\overline{\text{IOB}}$ = 0) or with memory and I/O ($\overline{\text{IOB}}$ = 1).

$\overline{\text{LOCK}}$ (Lock Input)—prevents the 8289 from allowing any other microprocessor to gain access to the shared bus. An 8086/8088 instruction that contains a LOCK prefix will prevent other microprocessors from accessing the shared bus.

RESB (Resident-Bus Input)—is a strapping connection that allows the 8289 to operate in systems that have either a shared-bus or resident-bus system. If RESB is a logic 1, the 8289 is configured as a shared-bus master. If RESB is a logic 0, the 8289 is configured as a local-bus master. When configured as a shared-bus master, access is requested through the SYSB/$\overline{\text{RESB}}$ input pin.

$\overline{\text{S2}}$, $\overline{\text{S1}}$, and $\overline{\text{S0}}$ (Status Inputs)—initiate shared-bus requests and surrenders. These pins connect to the 8288 system bus controller status pins.

SYSB/$\overline{\text{RESB}}$ (System Bus/Resident Bus Input)—selects the shared-bus system when placed at a logic 1, or the resident local bus when placed at a logic 0.

*General 8289 Operation.* As the pin descriptions demonstrate, the 8289 can be operated in three basic modes: (1) I/O peripheral-bus mode, (2) resident-bus mode, and (3) single-bus mode. Refer to Table 13–1 for the connections required to operate the 8289 in these modes. In the *I/O peripheral-bus mode*, all devices on the local bus are treated as I/O, including memory, and are accessed by I/I instructions. All memory references access the shared bus and all I/O access the resident (local) bus. The *resident-bus mode* allows memory and I/O accesses on both the local and shared buses. Finally, the *single-bus mode* interfaces a microprocessor to a shared bus, but the microprocessor has no local memory or local I/O. In many systems, one microprocessor is set up as the shared-bus master (single-bus mode) to control the shared bus and become the shared-bus master. The *shared-bus master* controls the system through shared memory and I/O. Additional microprocessors are connected to the shared bus as resident or I/O peripheral-bus masters. These additional bus masters usually perform independent tasks that are reported to the shared-bus master through the shared bus.

**TABLE 13–1** 8289 modes of operation

| *Mode* | *Pin Connections* |
|---|---|
| Single bus | $\overline{\text{IOB}}$ = 1 and RESB = 0 |
| Resident bus | $\overline{\text{IOB}}$ = 1 and RESB = 1 |
| I/O bus | $\overline{\text{IOB}}$ = 0 and RESB = 0 |
| I/O bus and resident bus | $\overline{\text{IOB}}$ = 0 and RESB = 1 |

*Single-Bus and Resident-Bus Connections.* Single-bus operation interfaces a microprocessor to a shared bus that contains both I/O and memory resources that are shared by other microprocessors. Figure 13–17 illustrates three 8088 microprocessors, each connected to a shared bus. Two of the three microprocessors operate in the resident-bus mode, while the third operates in the single-bus mode. Microprocessor A, in Figure 13–17, operates in the single-bus mode and has no local bus. This microprocessor only accesses the shared memory and I/O space. Microprocessor A is often referred to as the *system bus master* because it is responsible for coordinating the main memory and I/O tasks. The remaining two microprocessors (B and C) are connected in the resident-bus mode, which allows them access to both the shared bus and their own local buses. These resident-bus microprocessors are used to perform tasks that are independent from the system bus master. In fact, the only time the system bus master is interrupted from performing its tasks is when one of the two resident-bus microprocessors needs to transfer data between itself and the shared bus. This connection allows all three microprocessors to perform tasks simultaneously, yet data can be shared between microprocessors when needed.

In Figure 13–17, the bus master (Microprocessor A) allows the user to operate with a video terminal that allows the execution of programs and generally controls the system. Microprocessor B handles all telephone communications and passes this information to the shared memory in blocks. This means that Microprocessor B waits for each character to be transmitted or received and controls the protocol used for the transfers. For example, suppose that a 1K-byte block of data is transmitted across the telephone interface at the rate of 100 characters per second. This means that the transfer requires 10 seconds. Rather than tie up the bus master for 10 seconds, Microprocessor B patiently performs the data transfer from its own local memory and the local communications interface. This frees the bus master for other tasks. The only time Microprocessor B interrupts the bus master is to transfer data between the shared memory and its local memory system. This requires only a few hundred microseconds from the bus master and the main system.

Microprocessor C is used as a print spooler. Its only task is to print data on the printer. Whenever the bus master requires printed output, it transfers the task to Microprocessor C. Microprocessor C then accesses the shared memory, captures the data to be printed, and stores it in its own local memory. Data are then printed from the local memory, freeing the bus master to perform other tasks. This allows the system to execute a program with the bus master, transfer data through the communications interface with Microprocessor B, and print information on the printer with Microprocessor C. These tasks all execute simultaneously. There is no limit to the number of microprocessors connected to a system, or the number of tasks performed simultaneously using this technique. The only limit is that introduced by the system design and the designer's ingenuity.

## Priority Logic Using the 8289

In applications that use the 8289, there is always more than one microprocessor connected to a shared bus. Because only one can access the shared bus at a time, some method of resolving priority must be employed. There are two methods for resolving priority with the 8289 bus arbiter: the daisy-chain (serial) and the parallel-priority schemes.

*Daisy-Chain Priority.* The daisy-chain priority scheme connects the $\overline{\text{BPRO}}$ output to the $\overline{\text{BPRN}}$ input of the next-lower priority 8289 and is the least expensive priority scheme to

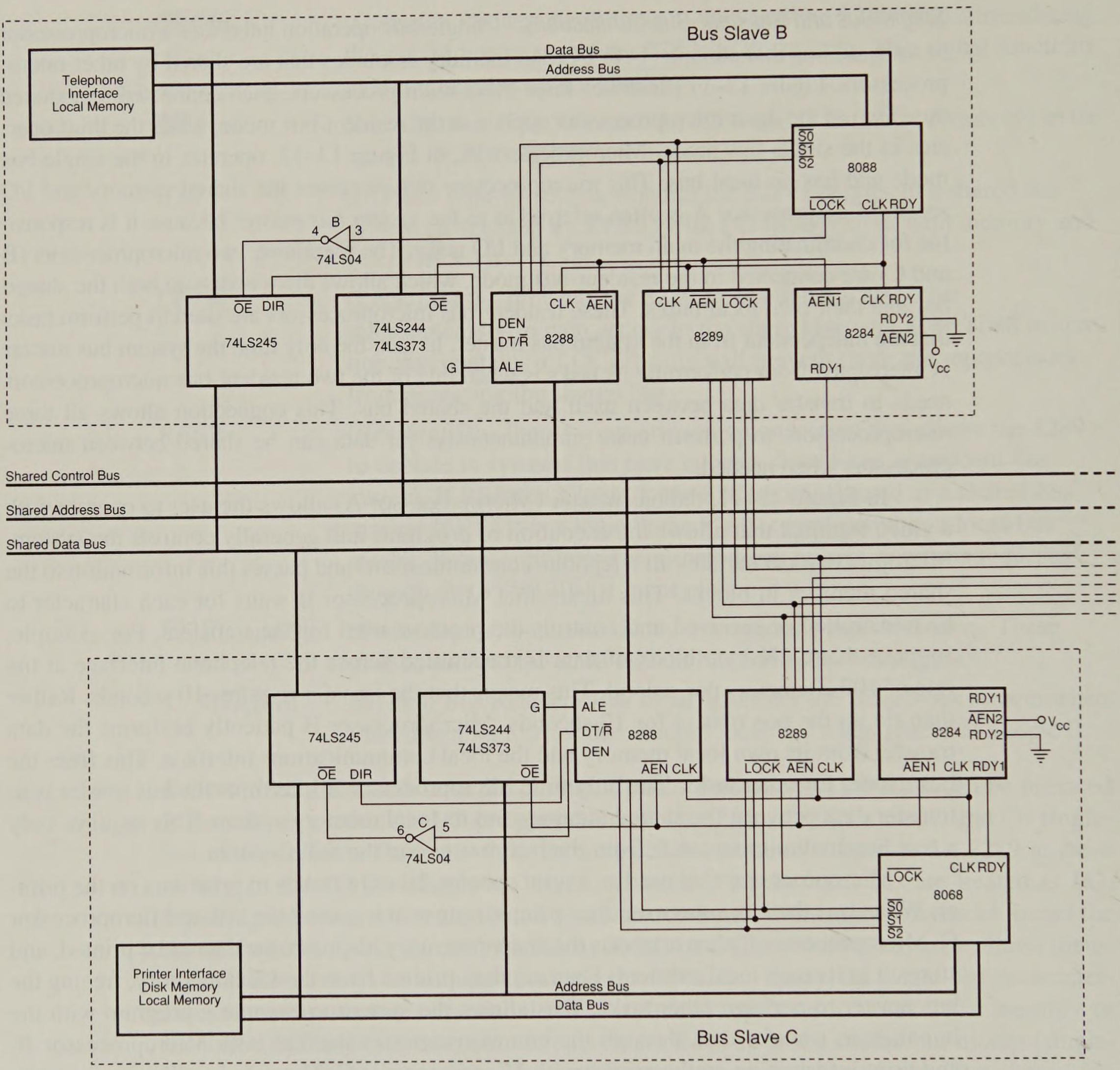



**FIGURE 13–17** Three 8088 microprocessors that share a common bus system. Microprocessor A is the bus master in control of the shared memory and CRT terminal. Microprocessor B is a bus slave controlling its local telephone interface and memory. Microprocessor C is also a slave that controls a printer, disk memory system, and local memory.

implement. Figure 13–18 illustrates the daisy-chain scheme for connecting several 8289s in a system. Because the $\overline{\text{BPRN}}$ input of the higher priority 8289 is grounded, it gets an immediate acknowledgment whenever its microprocessor requests access to the shared bus. The $\overline{\text{BPRO}}$ output is a logic 0 if the 8289 and its microprocessor are inactive, and a logic 1 if the 8289 and its microprocessor are actively using the shared bus. If no requests are active, all $\overline{\text{BPRN}}$ inputs will see a logic 0. As soon as the highest priority 8289 receives

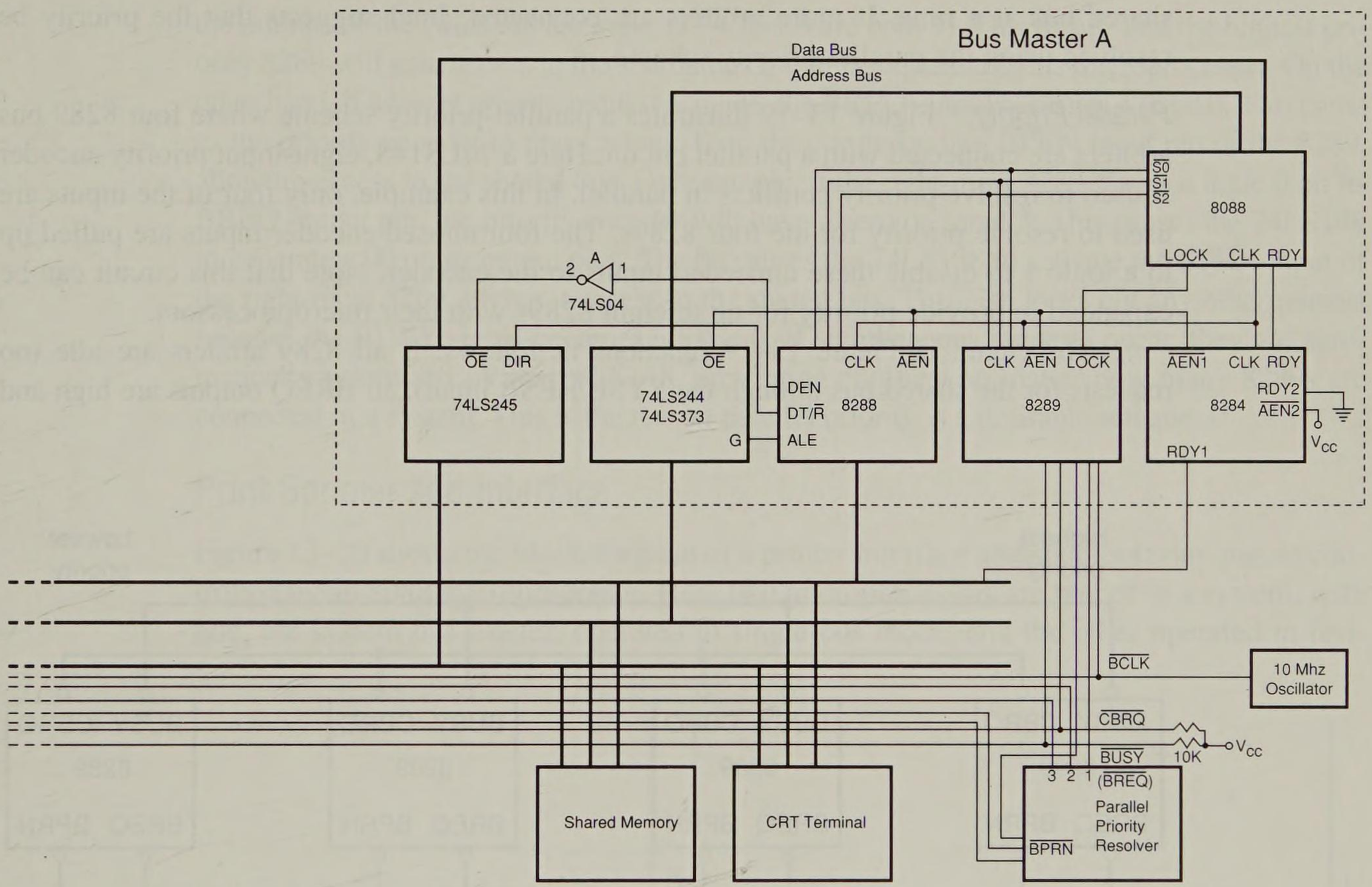



**FIGURE 13–17** (Continued)

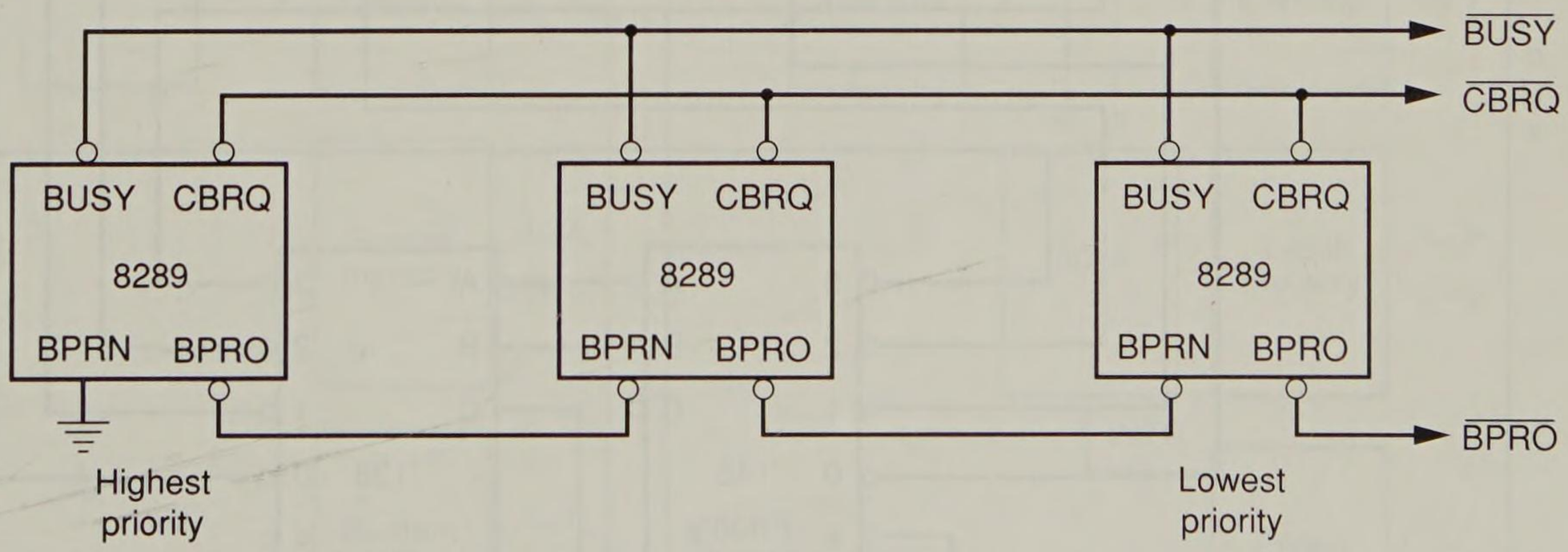



**FIGURE 13–18** Daisy-chain 8289 priority resolver.

a bus acknowledgment, its $\overline{\text{BPRO}}$ output goes high, blocking all lower priority 8289s from accessing the shared bus. If more than one 8289 receives an acknowledgment, more than one 8289 will function at the same time. This scheme is seldom used because two microprocessors can access the shared bus at a time. Intel recommends that this scheme be limited to no more than three 8289s in a system that uses a bus clock of 10 MHz or less. With this frequency bus clock, no conflict occurs and only one microprocessor accesses the

shared bus at a time. If more arbiters are connected, Intel suggests that the priority be resolved using the parallel-priority scheme.

*Parallel Priority.* Figure 13–19 illustrates a parallel-priority scheme where four 8289 bus arbiters are connected with a parallel circuit. Here a 74LS148, eight-input priority encoder is used to resolve priority conflicts in parallel. In this example, only four of the inputs are used to resolve priority for the four 8289s. The four unused encoder inputs are pulled up to a logic 1 to disable these unneeded inputs to the encoder. Note that this circuit can be expanded to provide priority for up to eight 8289s with their microprocessors.

The circuit in Figure 13–19 functions as follows. If all 8289 arbiters are idle (no requests for the shared bus through the SYSB/$\overline{\text{RESB}}$ input), all $\overline{\text{BREQ}}$ outputs are high and

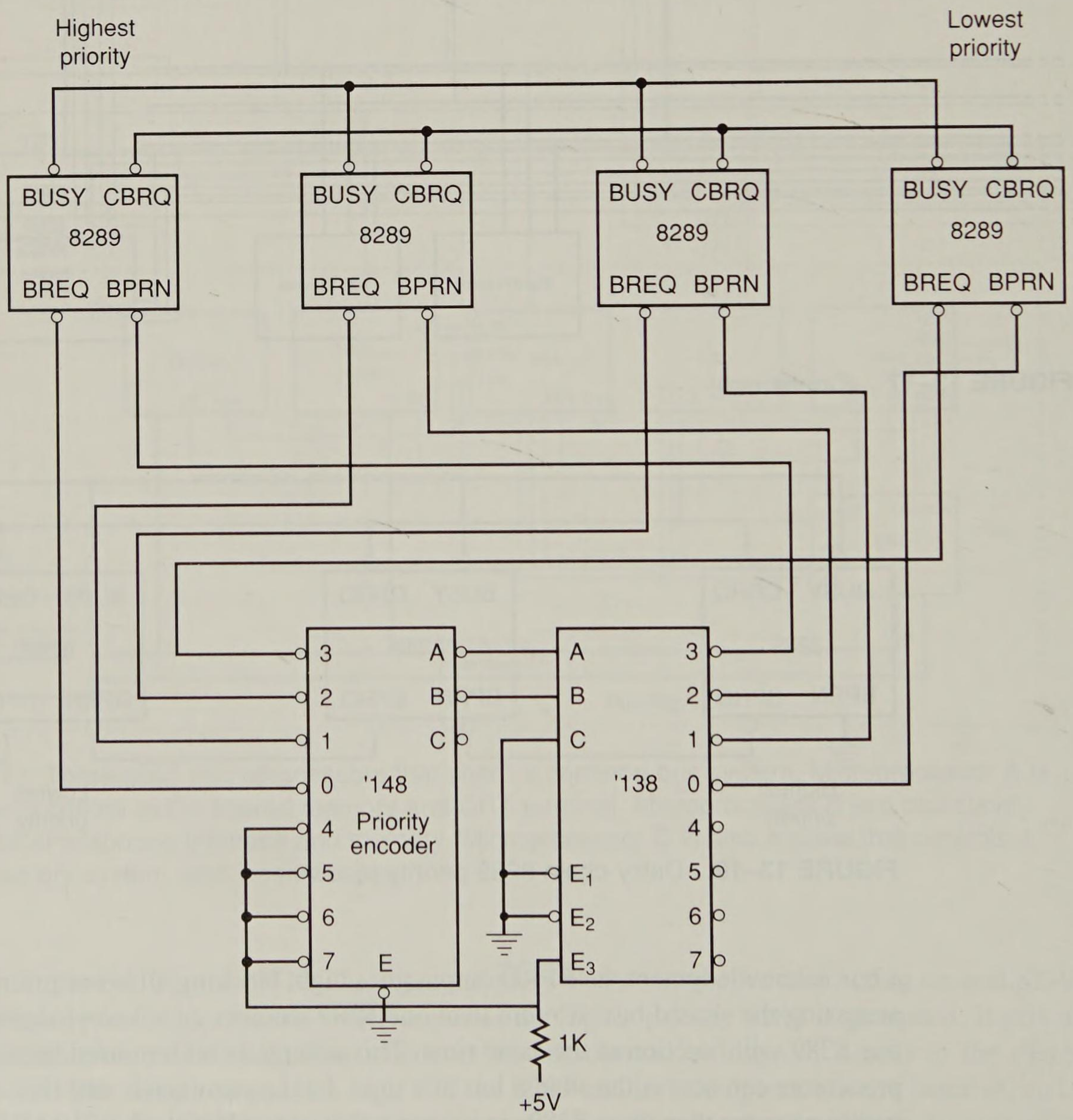



**FIGURE 13–19** Parallel-priority resolver for the 8289.

the outputs of the 74LS148 are logic 1s ($\overline{A}$ and $\overline{B}$ are both 1). This means that the highest priority 8289 will gain access to the shared bus if that is requested by its microprocessor. On the other hand, if a lower priority request is made, the $\overline{BREQ}$ output becomes a logic 0. This causes the priority encoder to place a logic 0 on the corresponding $\overline{BPRN}$ input pin of the 8289, allowing access to the shared bus. For example, if the right-most 8289 places a logic 0 on its $\overline{BREQ}$ output pin, the priority encoder will have a zero on input $\overline{3}$. This causes the 74LS148 to generate a 00 on its output pins. The 00 causes the 74LS138 to activate the $\overline{BPRN}$ input of the right-most 8289, giving it access to the shared bus. This also locks out any other request because the BUSY signal becomes a logic 0. If simultaneous requests occur, they are automatically prioritized by the 74LS148, preventing conflicts no matter how many 8289s are connected in a system. This is the reason parallel priority is a desirable scheme.

## Print Spooler and Interface

Figure 13–20 shows the block diagram of a printer interface and spooler (print queue) controlled by an 8088 microprocessor. Here two microprocessors are placed in a system, with one, the system bus master, operated in single-bus mode, and the other operated in resi-

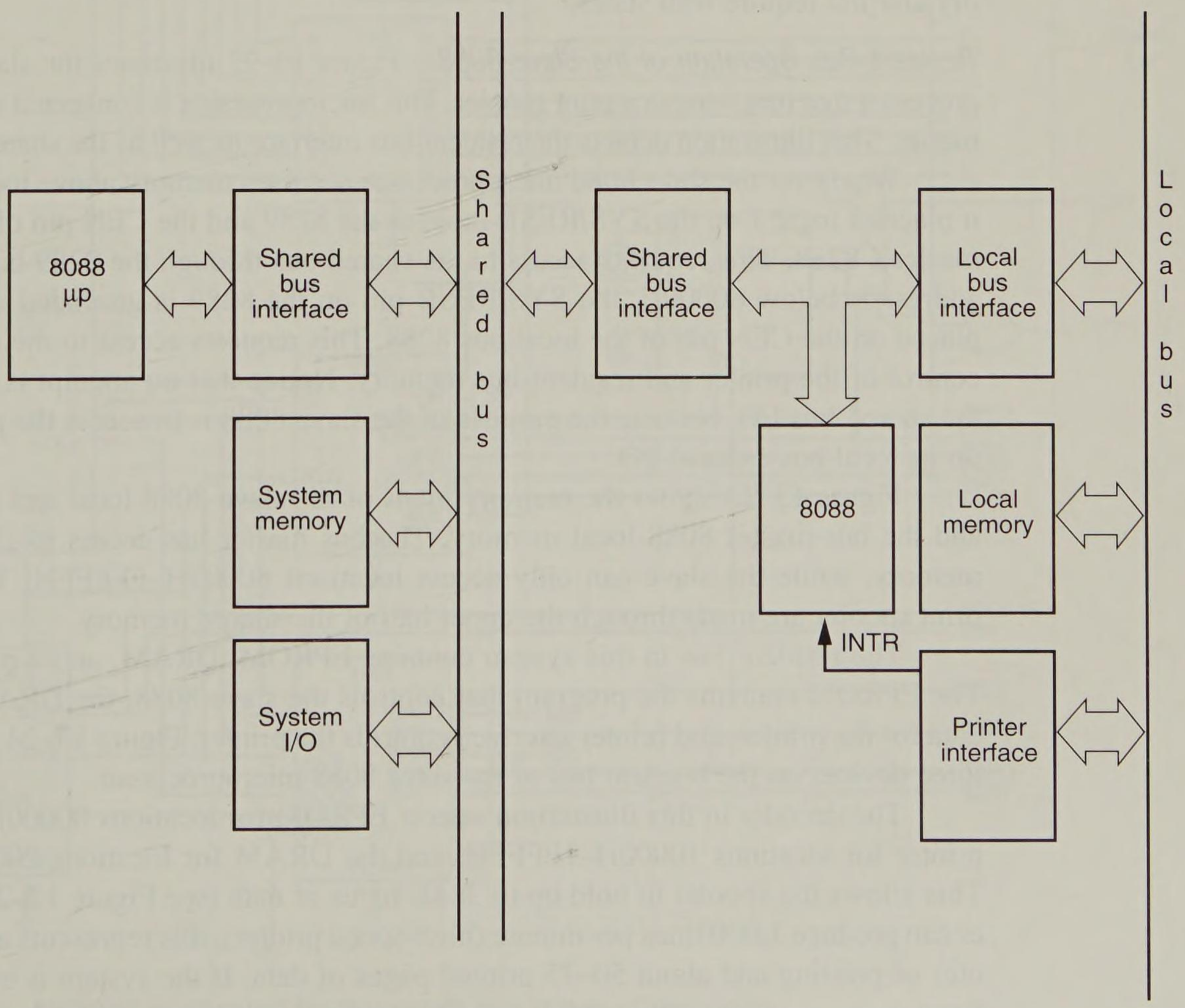



**FIGURE 13–20** Block diagram of two 8088s used to control a printer interface and print spooler.

dent-bus mode. Because two microprocessors exist, one can print data while the other is used to process new information.

In this interface, the slave microprocessor transfers data to the printer from its local memory without intervention from the system bus master. Data are transferred to the local memory of the slave microprocessor whenever it accesses the shared memory for additional data.

*Single Mode Bus Master Interface.* Figure 13–21 illustrates the 8088 bus master, operated in single mode, interfaced to the shared bus. The shared-bus master has access to every memory location and I/O device on the shared bus. The $\overline{\text{BCLK}}$ signal is generated by an 8284A clock generator used as a 10 MHz oscillator.

The 8289 arbiter is operated with no I/O bus, no resident bus, and ANYRQST pulled high, so the shared bus can be accessed for all memory and I/O transfers. This system uses the daisy-chain priority scheme, which means the $\overline{\text{BREQ}}$ signal is not connected. This interface allows the slave microprocessor to access shared memory whenever necessary and thus prevents the slave microprocessor from locking the bus for too long.

In addition to the normal bus signals, this circuit also provides the shared bus with a 2.5 MHz PCLK signal for any I/O device. The RDY input is shown in case system memory and I/O require wait states.

*Resident-Bus Operation of the Slave 8088.* Figure 13–22 illustrates the slave 8088 microprocessor that functions as a print spooler. This microprocessor is connected as a resident-bus master. This illustration depicts the resident-bus interface as well as the shared-bus interface.

Whenever the slave 8088 microprocessor accesses memory above location 7FFFFH, it places a logic 1 on the SYS/$\overline{\text{RESB}}$ input to the 8289 and the CEN pin of the shared-bus master's 8288. This requests access to the shared bus through the 8289 bus arbiter. If the address is below 80000H, the SYS/$\overline{\text{RESB}}$ pin on the 8289 is grounded and a logic 1 is placed on the CEN pin of the local bus 8288. This requests access to the resident bus for control of the printer and resident-bus memory. Notice that no attempt is made to access the shared-bus I/O, because the purpose of the slave 8088 is to access the printer interface on its local bus as local I/O.

Figure 13–23 shows the memory maps of the slave 8088 local and shared memory and the bus-master 8088 local memory. The bus master has access to all of the shared memory, while the slave can only access locations 80000H–FFFFFH. Transfers to the print spooler are made through the upper half of the shared memory.

The resident bus in this system contains EPROM, DRAM, and a printer interface. The EPROM contains the program that controls the slave 8088, the DRAM contains the data for the printer, and printer interface controls the printer. Figure 13–24 illustrates these three devices on the resident bus of the slave 8088 microprocessor.

The decoder in this illustration selects EPROM for locations 00000H–0FFFFH, the printer for locations 10000H–1FFFFH, and the DRAM for locations 20000H–7FFFFH. This allows the spooler to hold up to 384K bytes of data (see Figure 13–25). If the printer can produce 1,000 lines per minute (high-speed printer), this represents about three minutes of printing and about 50–75 printed pages of data. If the system is expanded with a fixed memory system, the capacity of the spooler can become almost boundless because of the huge capacity of a modern fixed-disk memory system. For simplicity's sake, this is not added to this example.

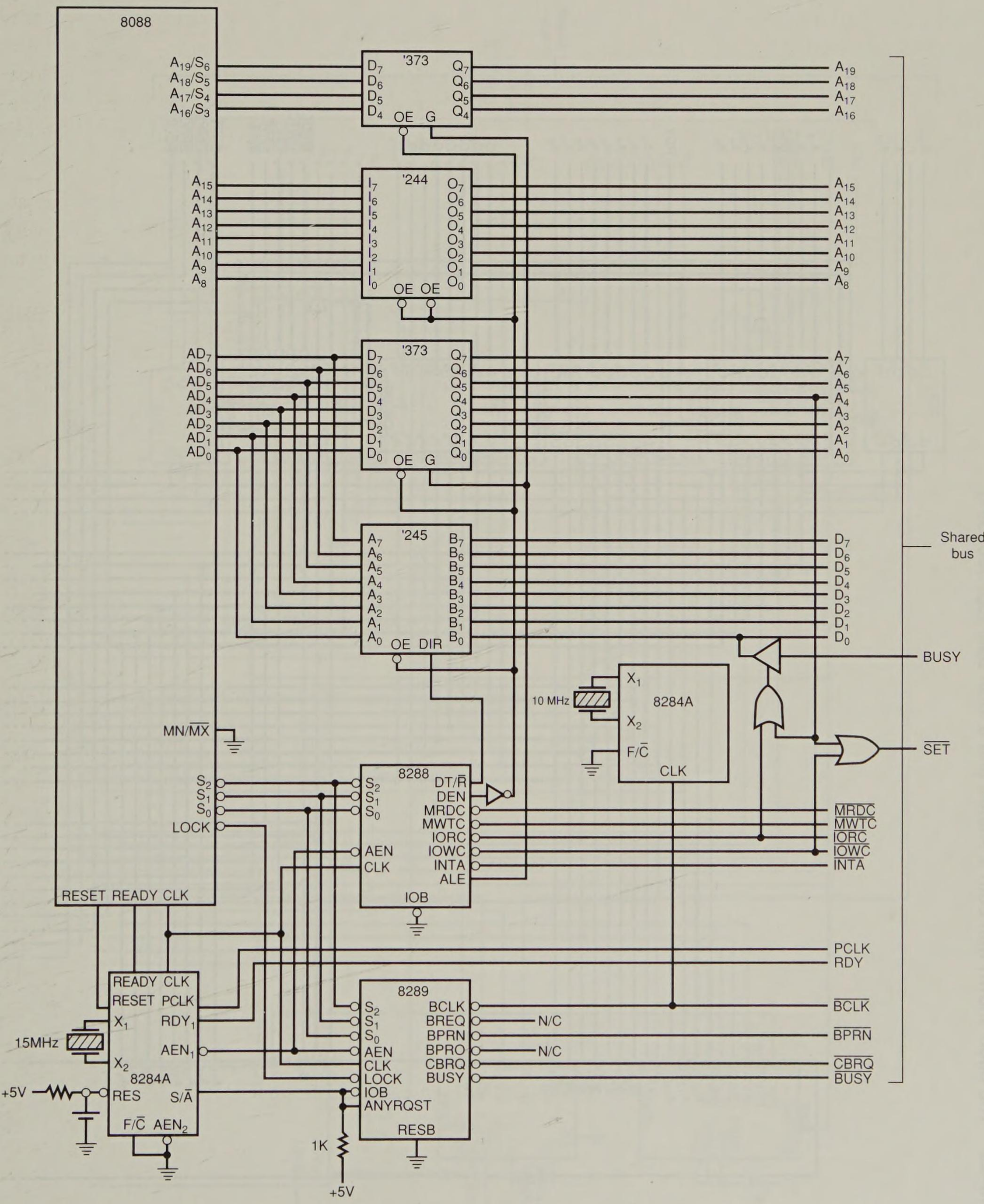
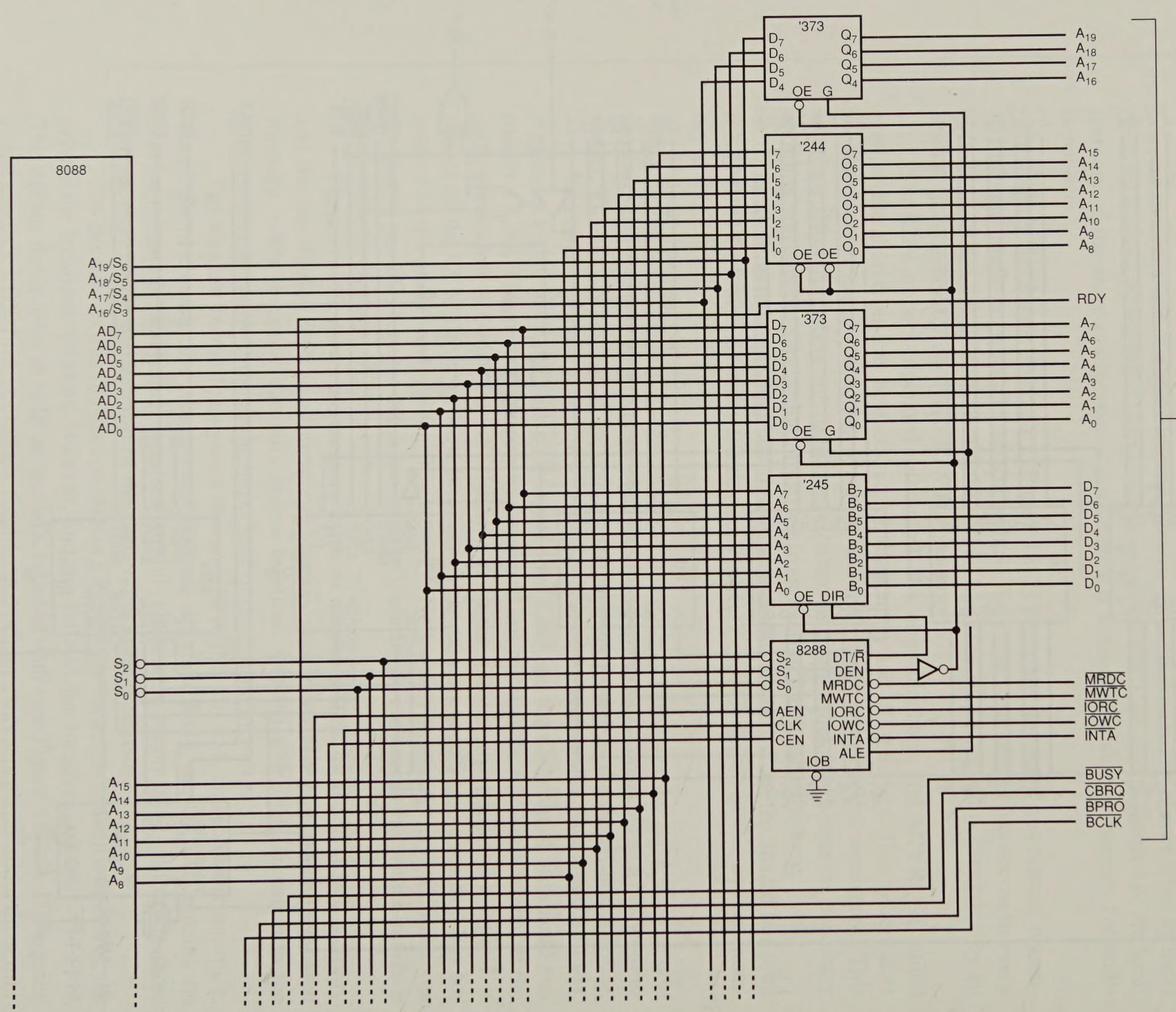
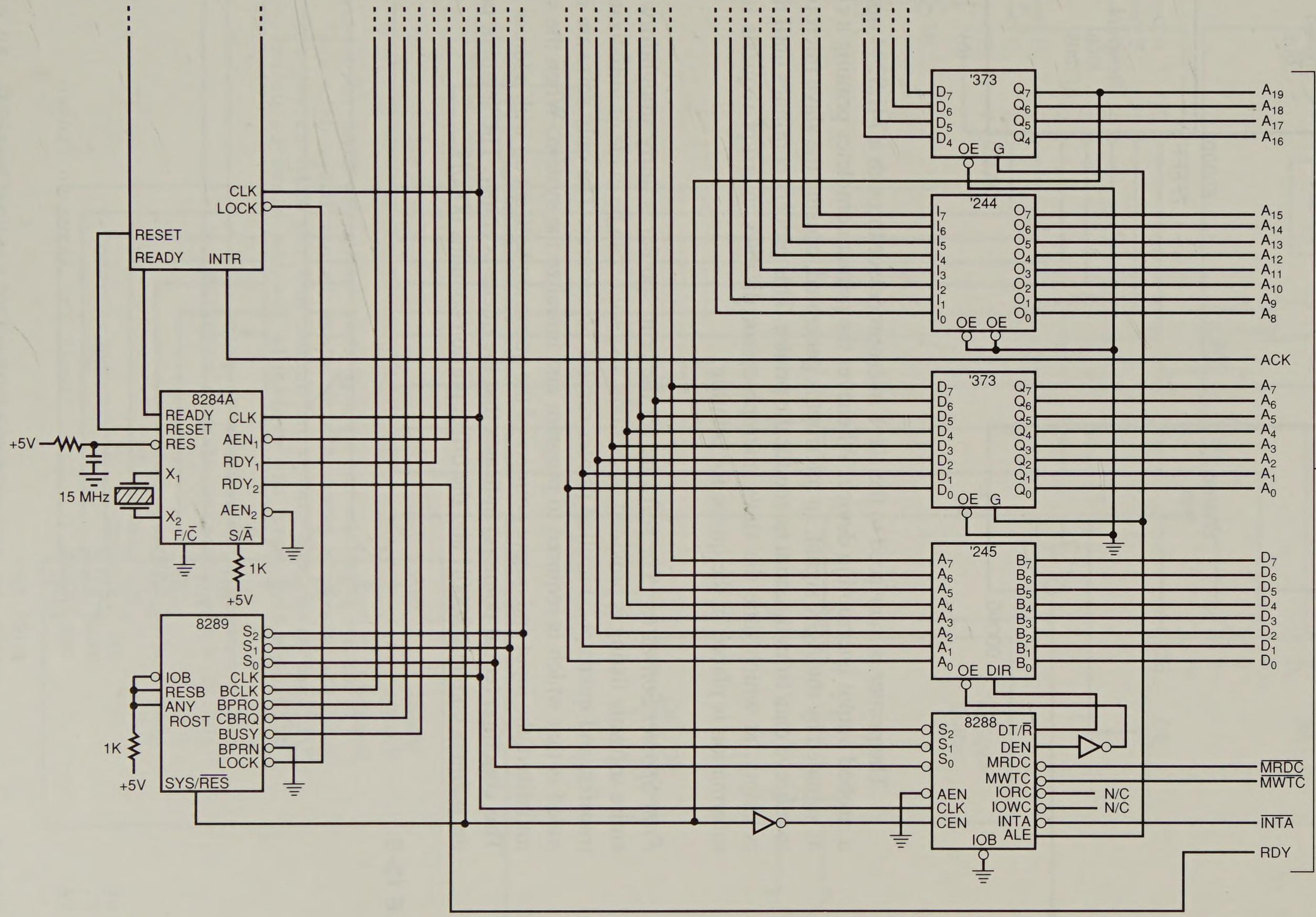



**FIGURE 13–21** An 8088 connected for the single-bus mode of operation.

**FIGURE 13–22** The 8088 shown with both a shared and a local bus. (pp. 590–591)




CLK
LOCK
RESET
READY
INTR
8284A
READY
RESET
RES
CLK
AEN1
RDY1
RDY2
AEN2
X1
X2
F/C
S/A
+5V
15 MHz
1K
+5V
8289
S2
S1
S0
IOB
RESB
ANY
ROST
CLK
BCLK
BPRO
CBRQ
BUSY
BPRN
LOCK
SYS/RES
1K
+5V
'373
D7
D6
D5
D4
OE
G
Q7
Q6
Q5
Q4
A19
A18
A17
A16
'244
I7
I6
I5
I4
I3
I2
I1
I0
OE
OE
O7
O6
O5
O4
O3
O2
O1
O0
A15
A14
A13
A12
A11
A10
A9
A8
ACK
'373
D7
D6
D5
D4
D3
D2
D1
D0
OE
G
Q7
Q6
Q5
Q4
Q3
Q2
Q1
Q0
A7
A6
A5
A4
A3
A2
A1
A0
Resident local bus
'245
A7
A6
A5
A4
A3
A2
A1
A0
OE
DIR
B7
B6
B5
B4
B3
B2
B1
B0
D7
D6
D5
D4
D3
D2
D1
D0
8288
S2
S1
S0
AEN
CLK
CEN
DT/R
DEN
MRDC
MWTC
IORC
IOWC
INTA
ALE
IOB
MRDC
MWTC
N/C
N/C
INTA
RDY

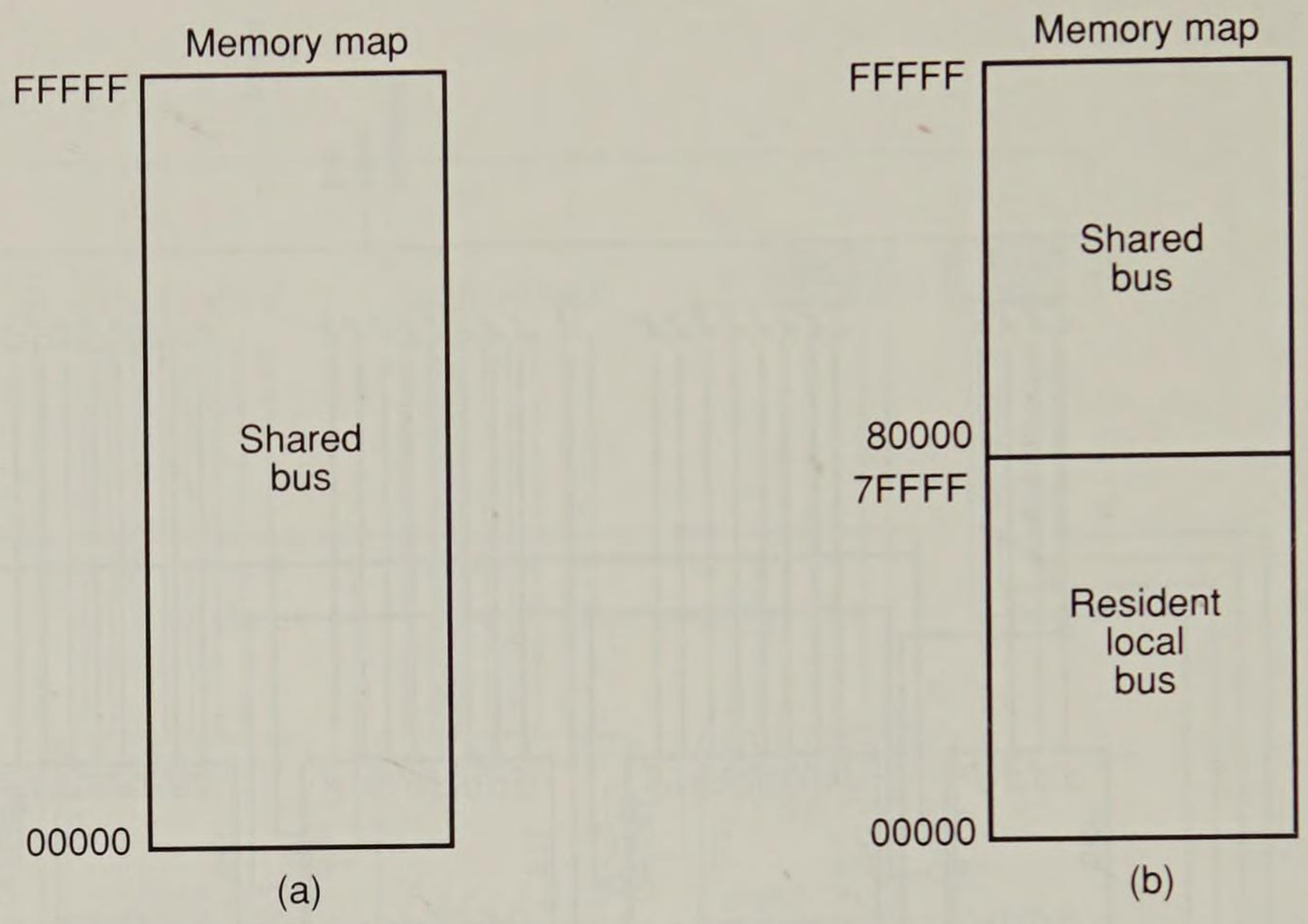



**FIGURE 13–23** Memory maps for the print spooler. (a) Shared-bus master. (b) Bus-slave memory map.

The printer is interfaced to the slave microprocessor through a 74LS244 operated as a strobed output interrupting device. Whenever the printer completes printing a character, as signaled by the $\overline{\text{ACK}}$ signal, an interrupt is generated, causing the slave microprocessor to retrieve data from the print queue for the printer. This process continues until the queue is empty, at which time the slave microprocessor disables interrupt requests until new information is placed in the queue for printing.

*Print Spooler Software.* The software for the print spooler is fairly straightforward. The entire software listing is provided in sections, so a study can be made of initialization, data transfer, and interrupt-controlled printer service procedures. The only software not illustrated is that which is required to program and initialize the system. When the system is initialized, the input and output pointers of the queue are both set up with address 20000H. This condition (equal pointers) indicates that the queue is empty. The segment portion of the pointers contains 2000H and the offset portion contains 0000H.

**EXAMPLE 13–5**

```
                        ;Procedure found in the master that tests the FLAG to determine
                        ;if the slave is busy. If the slave is not busy, data are
                        ;transferred into the print buffer and the FLAG is set to
                        ;indicate that data are available for the slave to print.
                        ;
                        ;CX = number of characters to print.
                        ;DS:SI = the location of data to print.
                        ;
0000                    LOAD    PROC FAR

0000 E4 00                      IN    AL,00H
0002 A8 01                      TEST  AL,1
0004 75 FA                      JNZ   LOAD              ;wait for slave to finish

0006 06                         PUSH  ES
0007 BF 0002 R                  MOV   DI,OFFSET BUFFER
```

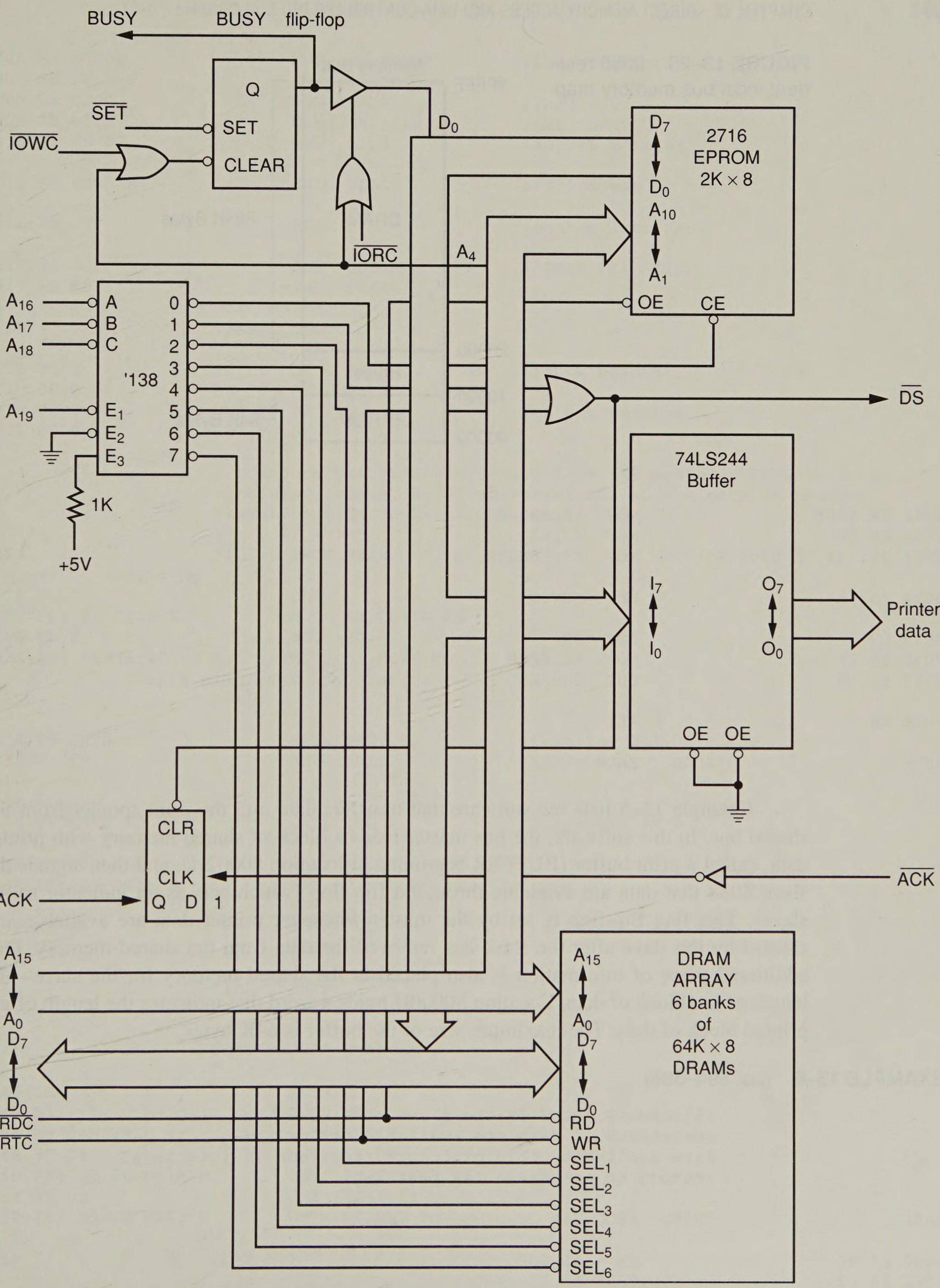
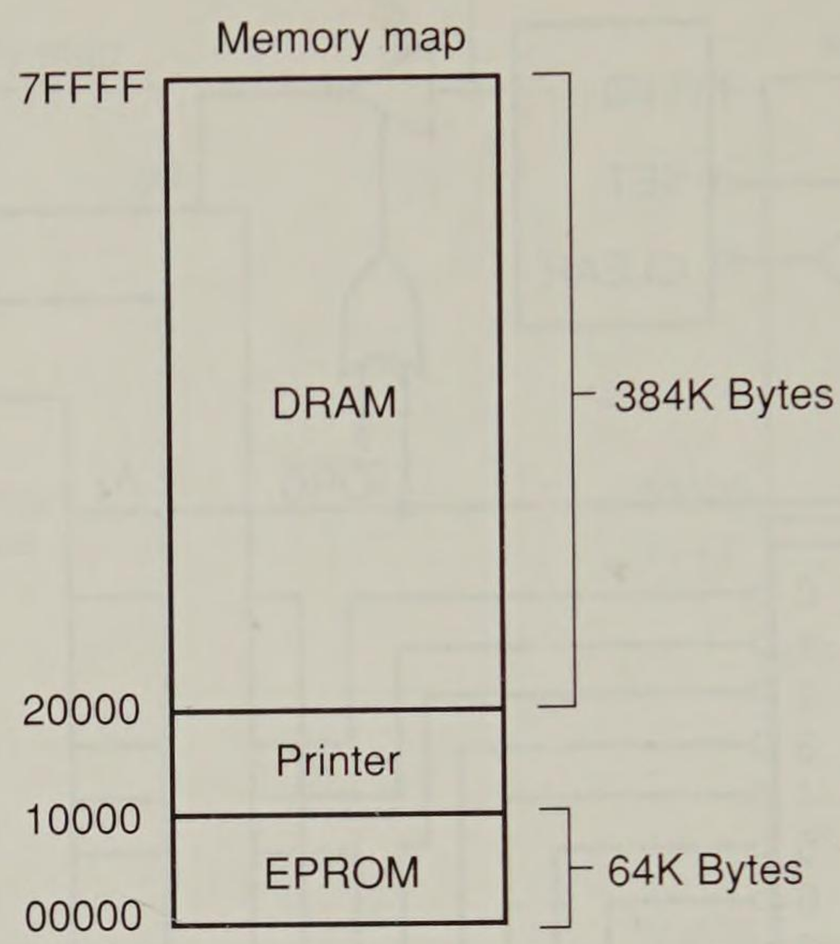



**FIGURE 13–24** Resident local bus for the print spooler.

**FIGURE 13–25** 8088 resident local bus memory map.




```
000A B8 8000                    MOV   AX,8000H
000D 8E C0                      MOV   ES,AX
000F 26: 89 0E 0000 R           MOV   ES:LENGTH,CX

0014 F3/ A4                     REP   MOVSB

0016 07                         POP   ES
0017 B0 FF                      MOV   AL,0FFH
0019 E6 00                      OUT   00H,AL          ;set data available flag

001B CB                         RET

001C                    LOAD    ENDP
```

Example 13–5 lists the software that transfers data into the print spooler from the shared bus. In this software, the bus master loads a block of shared memory with printer data, called a print buffer (BUFFER beginning at location 80002H), and then signals the slave 8088 that data are available through a flip-flop (which acts as an indicator to the slave). This flag flip-flop is set by the master whenever printer data are available and cleared by the slave after the slave has removed the data from the shared memory. One additional piece of information is also placed in the shared memory for the slave—the length of the block of data. Location 80000H holds a word that indicates the length of the printed block of data. The maximum size of the buffer is 64K bytes.

**EXAMPLE 13–6** (pp. 594–596)

```
                        ;Procedure found in the slave that tests the FLAG to
                        ;determine if data are available for printing.  If data
                        ;are available, this procedure transfers it into local
                        ;memory to the slave for printing.
                        ;
0000                    TRANS  PROC FAR

0000 E4 00                      IN    AL,00H
0002 A8 01                      TEST  AL,1
0004 74 FA                      JZ    TRANS           ;if no data
```

```
0006 B8 8000                    MOV  AX,8000H          ;load segment
0009 8E D8                      MOV  DS,AX
000B BE 0000                    MOV  SI,0              ;load source
000E 8B 0C                      MOV  CX,[SI]           ;get count
0010 83 C6 02                   ADD  SI,2              ;adjust pointer

0013 E8 0023 R                  CALL FILL              ;fill queue

0016 FB                         STI                    ;enable interrupt

0017 E6 00                      OUT  00H,AL            ;clear flip-flop
0019 EB E5                      JMP  TRANS

001B                    TRANS   ENDP

001B 0000               IN_PS   DW   ?                 ;input pointer
001D 0000               IN_PO   DW   ?
001F 0000               OUT_PS  DW   ?                 ;output pointer
0021 0000               OUT_PO  DW   ?

                        ;Procedure that transfers data from the master BUFFER to
                        ;the slave queue. DS:SI addresses the source data in shared
                        ;memory and CX contains the count.
                        ;
0023                    FILL    PROC NEAR

0023 FC                         CLD
0024 2E: A1 001B R              MOV  AX,CS:IN_PS
0028 8E C0                      MOV  ES,AX
002A 2E: 8B 3E 001DH R          MOV  DI,C:IN_PO

002F                    FULL:

002F E8 0036 R                  CALL TEST_F            ;test for full
0032 74 FB                      JZ   FULL              ;if full

0034 A4                         MOVSB
0035 E8 005C R                  CALL INC_P             ;increment IN_P

0038 8C C0                      MOV  AX,ES
003A 2E: A3 001B R              MOV  CS:IN_PS,AX
003E 2E: 89 3E 001D R           MOV  CS:IN_PO,DI
0043 E2 EA                      LOOP FULL              ;repeat for all data
0045 C3                         RET

0046                    FILL    ENDP
                        ;Procedure to test for a full queue
                        ;
0046                    TEST_F  PROC NEAR

0046 1E                         PUSH DS
0047 57                         PUSH DI
0048 E8 005C R                  CALL INC_P
004B 8C D8                      MOV  AX,DS
004D 2E: 3B 06 001F R           CMP  AX,CS:OUT_PS
0052 75 05                      JNE  TEST_F1
0054 2E: 3B 3E 0021 R           CMP  DI,OUT_PO

0059                    TEST_F1:

0059 5F                         POP  DI
```

```
005A 1F                         POP  DS
005B C3                         RET

005C                    TEST_F ENDP

                        ;Procedure to increment the input pointer.
                        ;
005C                    INC_P  PROC NEAR

005C 47                         INC  DI
005D 0B FF                      OR   DI,DI
005F 75 0F                      JNE  INC_PE
0061 8C C0                      MOV  AX,ES
0063 05 1000                    ADD  AX,1000H
0066 3D 8000                    CMP  AX,8000H
0069 75 03                      JNE  INC_P1
006B B8 2000                    MOV  AX,2000H

006E                    INC_P1:

006E 8E C0                      MOV  ES,AX

0070                    INC_PE:

0070 C3                         RET

0071                    INC_P  ENDP
```

Once the slave notices a logic 1 in the flag flip-flop it begins transferring data from the shared memory into its own local memory. The data are stored in the local memory organized as a FIFO (first-in, first out) queue. In this example, the size of the queue is 384K bytes. Example 13–6 lists the software used by the slave to load the queue from the shared memory buffer. In this software the slave tests the flag flip-flop to see whether the master has filled the buffer in the shared memory. If the buffer is filled, the slave microprocessor transfers its contents from the shared memory into its queue. Once the transfer is complete, the slave clears the flag flip-flop so the master may begin filling the buffer with additional printer data.

**EXAMPLE 13–7**

```
                        ;Interrupt service procedure that prints data from the queue.
                        ;If all data are printed, interrupts are disabled.
                        ;
0071                    PRINT  PROC FAR

0071 50                         PUSH AX                 ;save registers
0072 55                         PUSH BP
0073 1E                         PUSH DS
0074 57                         PUSH DI
0075 06                         PUSH ES

0076 E8 009D R                  CALL TEST_E
0079 75 0E                      JNE  PRINT1             ;if not empty

007B 8B EC                      MOV  BP,SP
007D 8B 46 0C                   MOV  AX,[BP+12]
0080 80 E4 FD                   AND  AH,0FDH            ;interrupt off
```

```
0083 89 46 0C                   MOV  [BP+12],AX
0086 EB 0F 90                   JMP  PRINTE

0089                    PRINT1:

0089 B8 1000                    MOV  AX,1000H
008C 8E C0                      MOV  ES,AX
008E 8A 05                      MOV  AL,[DI]
0090 26: A2 0000 R              MOV  ES:DATA,AL      ;print character

0094 E8 00B5 R                  CALL INC_O

0097                    PRINTE:

0097 07                         POP  ES              ;restore registers
0098 5F                         POP  DI
0099 1F                         POP  DS
009A 5D                         POP  BP
009B 58                         POP  AX
009C CF                         IRET

009D                    PRINT  ENDP

009D                    TEST_E PROC NEAR

009D 2E: A1 001F R              MOV  AX,CS:OUT_PS
00A1 8E D8                      MOV  DS,AX
00A3 2E: 8B 3E 0021 R           MOV  DI,CS:OUT_PO
00A8 2E: 3B 06 001B R           CMP  AX,CS:IN_PS
00AD 75 05                      JNE  TEST_EE
00AF 2E: 3B 3E 001D R           CMP  DI,CS:IN_PO

00B4                    TEST_EE:

00B4 C3                         RET

00B5                    TEST_E ENDP
00B5                    INC_O  PROC NEAR

00B5 47                         INC  DI
00B6 0B FF                      OR   DI,DI
00B8 75 0D                      JNE  INC_OE
00BA 8C C0                      MOV  AX,DS
00BC 05 1000                    ADD  AX,1000H
00BF 3D 8000                    CMP  AX,8000H
00C2 75 03                      JNE  INC_OE
00C4 B8 2000                    MOV  AX,2000H

00C7                    INC_OE:

00C7 2E: A3 001F R              MOV  CS:OUT_PS,AX
00CB 2E: 89 3E 0021 R           MOV  CS:OUT_PO,DI
00D0 C3                         RET

00D1                    INC_O  ENDP
```

Example 13–7 lists the software that prints data from the queue. This software is interrupt-driven, and therefore runs as a background program, virtually hidden from the software listed in Example 13–6 except for the interrupt enable instruction. Whenever the

printer interface, through a flip-flop, indicates that it is ready to accept additional data, an interrupt occurs, calling this interrupt service procedure. The procedure extracts data from the queue and sends them to the printer.

## 13–4 DISK MEMORY SYSTEMS

Disk memory is used to store long-term data. Many types of disk storage systems are available today. All disk memory systems use magnetic media except the optical disk memory, which stores data on a plastic disk. Optical disk memory is either a *CDROM* (compact disk/read only memory) that is read but never written, or a *WORM* (write once/read mostly) that is read most of the time but can be written once by a laser beam. Also becoming available is optical disk memory that can be read and written many times, but there is still a limitation on the number of write operations allowed. This section of the chapter provides an introduction to disk memory systems so that they may be used with computer systems. It also provides detail of their operation.

### Floppy Disk Memory

The most common and the most basic form of disk memory is the floppy or flexible disk. This magnetic recording medium is available in three sizes: the 8″ *standard*, 5¼″ *mini-floppy*, and the 3½″ *micro-floppy*. Today the 8″ standard version has all but disappeared, giving way to the mini- and micro-floppy disks. The 8″ disk is too large and unwieldy to handle and stockpile. To solve this problem, industry developed the 5¼″ mini-floppy disk. Today, the micro-floppy disk is quickly replacing the mini-floppy because of its reduced size, ease of storage, and durability. Even so, many systems are still marketed with the mini-floppy disk drive. In fact, one vendor markets a single disk drive that accepts both 5¼″ and 3½″ floppy disks.

All disks have several things in common. They are all organized so data are stored in tracks. A *track* is a concentric ring of data that is stored on a surface of a disk. Figure 13–26 illustrates the surface of a 5¼″ mini-floppy disk showing a track that is divided into sectors. A *sector* is a subdivision of track designed to hold a reasonable amount of data. In many systems, a sector holds either 512 or 1,024 bytes of data. The size of a sector can vary from 128 bytes to the length of one entire track.

Notice in Figure 13–26 that there is a hole through the disk, labeled an index hole. The *index hole* is designed so the electronic system that reads the disk is able to find the beginning of a track and its first sector (00). Tracks are numbered from track 00, the outer-most track, in increasing value toward the center, or inner-most, track. Sectors are often numbered from sector 00 on the outer-most track, to whatever value is required to reach the inner-most track and its last sector.

*The 5¼″ Mini-Floppy Disk.* Today, the 5¼″ floppy is probably the most popular disk size used with older microcomputer systems. Figure 13–27 illustrates this mini-floppy disk. The floppy disk is rotated at 300 RPMs inside its semi-rigid plastic jacket. The head mechanism in a floppy disk drive makes physical contact with the surface of the disk which

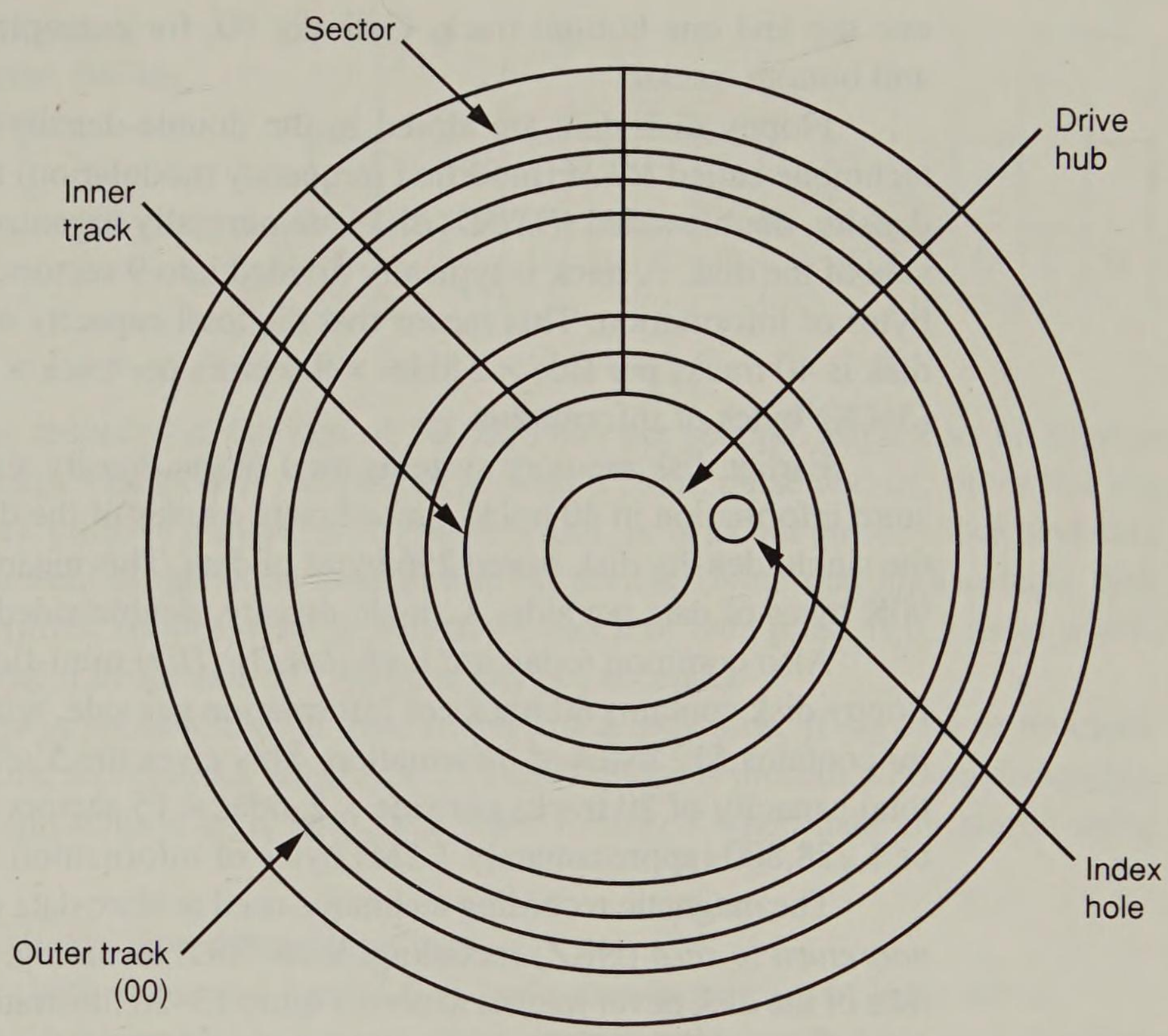



**FIGURE 13–26** The format of a 5¼″ floppy disk.

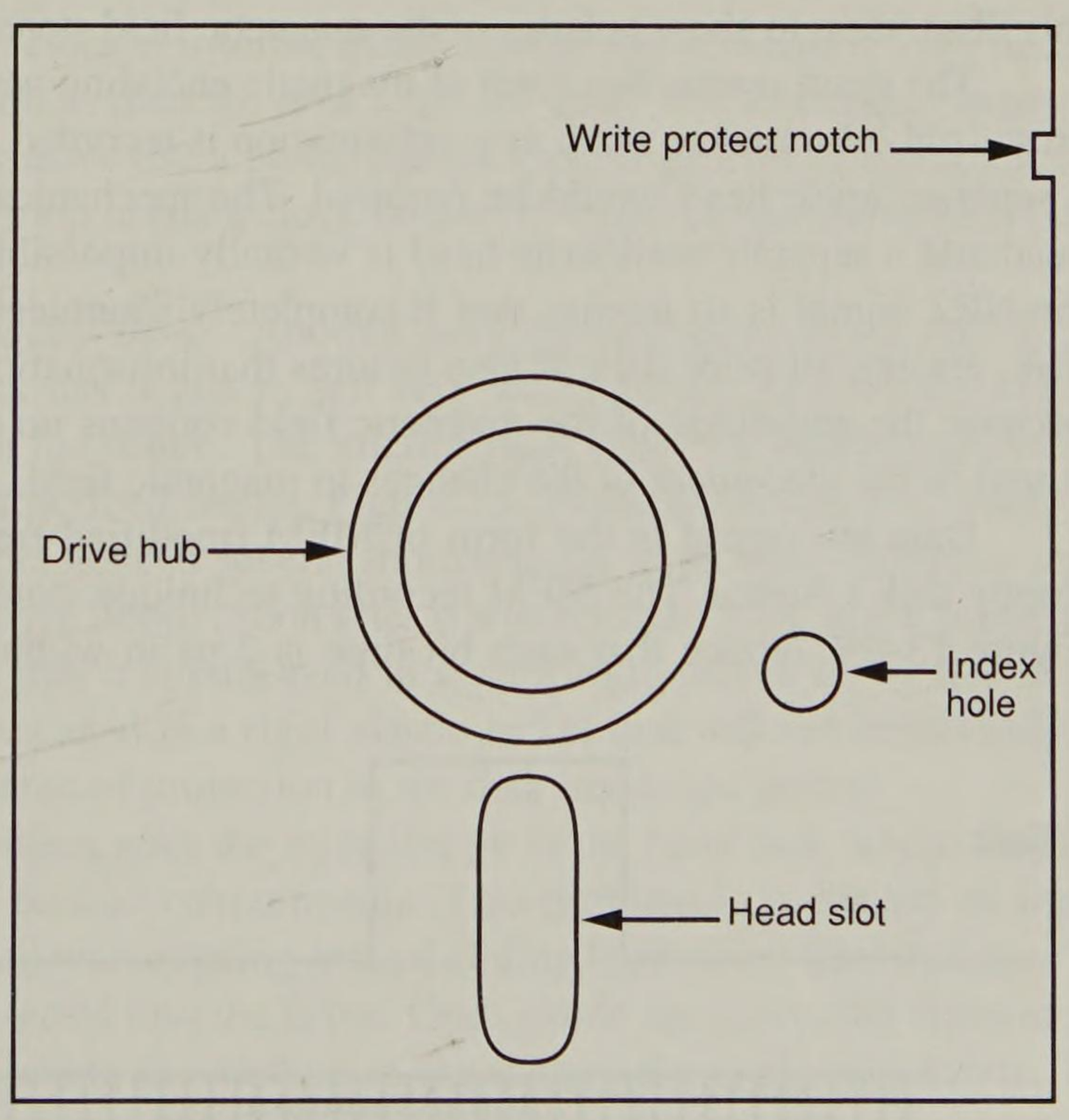



**FIGURE 13–27** The 5¼″ mini-floppy disk.

eventually causes wear and damage to the disk. This presents a potential for problems, but disks usually last many years before data is lost due to wear.

Today, most mini-floppy disks are double-sided. This means that data are written on both the top and bottom surfaces of the disk. A *cylinder* is a set of tracks consisting of

one top and one bottom track. Cylinder 00, for example, consists of the outer-most top and bottom tracks.

Floppy disk data are stored in the double-density format, which uses a recording technique called *MFM* (modified frequency modulation) to store the information. Double-density, double-sided (*DSDD*) disks are normally organized with 40 tracks of data on each side of the disk. A track is typically divided into 9 sectors, with each sector containing 512 bytes of information. This means that the total capacity of a double-density, double-sided disk is 40 tracks per side × 2 sides × 9 sectors per track × 512 bytes per sector, or 368,640 (360K) bytes of information.

Earlier disk memory systems used single-density and *FM* (frequency modulation) to store information in 40 tracks on one or two sides of the disk. Each of the 8 or 9 sectors on the single-density disk stored 256 bytes of data. This meant that a single-density disk stored 90K bytes of data per side. A single-density, double-sided disk stored 180K bytes of data.

Also common today are *high-density (HD)* mini-floppy disks. A high-density mini-floppy disk contains 80 tracks of information per side, with 15 sectors per track. Each sector contains 512 bytes of information. This gives the $5\frac{1}{4}''$ high-density mini-floppy disk a total capacity of 80 tracks per side × 2 sides × 15 sectors per track × 512 bytes per sector, or 1,228,800 (approximately 1.2M) bytes of information.

The magnetic recording technique used to store data on the surface of the disk is called *nonreturn to zero* (NRZ) recording. With NRZ recording, magnetic flux placed on the surface of the disk never returns to zero. Figure 13–28 illustrates the information stored in a portion of a track. It also shows how the magnetic field encodes the data. Arrows are used in this illustration to show polarity of the magnetic field stored on the surface of the disk.

The main reason this form of magnetic encoding was chosen is that it automatically erases old information when new information is recorded. If another technique were used, a separate erase head would be required. The mechanical alignment of a separate erase head and a separate read/write head is virtually impossible. The magnetic flux density of the NRZ signal is so intense that it completely saturates (magnetizes) the surface of the disk, erasing all prior data. It also ensures that information will not be affected by noise, because the amplitude of the magnetic field contains no information. The information is stored in the placement of the changes in magnetic field.

Data are stored in the form of MFM (modified frequency modulation) in modern floppy disk systems. The MFM recording technique stores data in the form illustrated in Figure 13–29. Notice that each bit time is 2 μs in width on a double-density disk. This

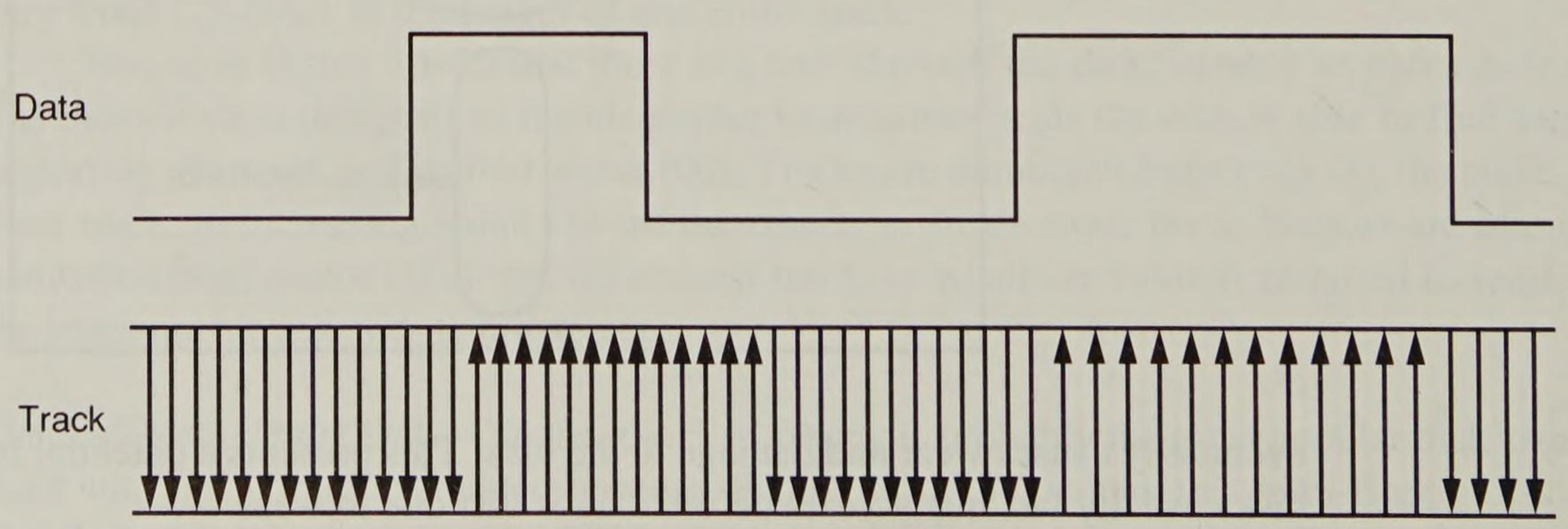



**FIGURE 13–28** The nonreturn to zero (NRZ) recording technique.

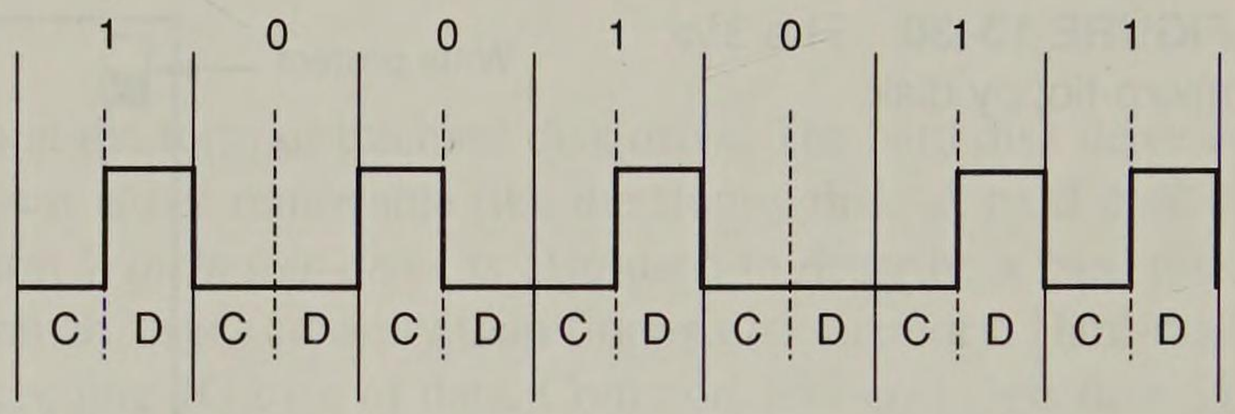



**FIGURE 13–29** Modified frequency modulation (MFM) used with disk memory.

means that data are recorded at the rate of 500,000 bits per second. Each 2 μs bit time is divided into two parts. One part is designated to hold a clock pulse and the other holds a data pulse. If a clock pulse is present, it is 1 μs in width, as is a data pulse. Clock and data pulses are never present at the same time in one bit period. (Note that high-density disk drives halve these times, so a bit time is 1 μs and a clock or data pulse is 0.5 μs in width. This also doubles the transfer rate to 1 million bits per second.)

If a data pulse is present, the bit time represents a logic one. If no data or no clock is present, the bit time represents a logic zero. If a clock pulse is present with no data pulse, the bit time also represents a logic zero. The rules followed when data are stored using MFM are:

1. A data pulse is always stored for a logic one.
2. No data and no clock are stored for the first logic zero in a string of logic zeros.
3. The second and subsequent logic zeros in a row contain a clock pulse, but no data pulse.

The reason a clock is inserted as the second and subsequent zero in a row is to maintain synchronization as data are read from the disk. The electronics used to recapture the data from the disk drive use a phase-locked loop to generate a clock and a data window. The phase-locked loop needs a clock or data to maintain synchronized operation.

*The 3½" Micro-Floppy Disk.* Another very popular disk size is the 3½" micro-floppy disk. This size disk has begun to sell very well recently; it promises to be the dominant size floppy disk in the future. The micro-floppy disk is a much-improved version of the mini-floppy disk described earlier. Figure 13–30 illustrates the 3½" micro-floppy disk.

Disk designers noticed several shortcomings of the mini-floppy, a scaled-down version of the 8" standard floppy, soon after it was released. One of the biggest problems with the mini-floppy is that it is packaged in a semi-rigid plastic cover that bends easily. The micro-floppy is packaged in a rigid plastic jacket that will not bend easily. This provides a much greater degree of protection to the disk inside the jacket.

Another problem with the mini-floppy is the head slot, which continually exposes the surface of the disk to contaminants. This problem is corrected on the micro-floppy, which is constructed with a spring-loaded sliding head door. The head door remains closed until the disk is inserted into the drive. Once inside the drive, the drive mechanism slides the door open, exposing the surface of the disk to the read/write heads. This provides a great deal of protection to the surface of the micro-floppy disk.

Another improvement is the sliding plastic write-protection mechanism on the micro-floppy disk. On the mini-floppy disk, a piece of tape was placed over a notch on the side of the jacket to prevent writing. This plastic tape easily became dislodged inside disk drives, causing problems. On the micro-floppy, an integrated plastic slide has replaced the

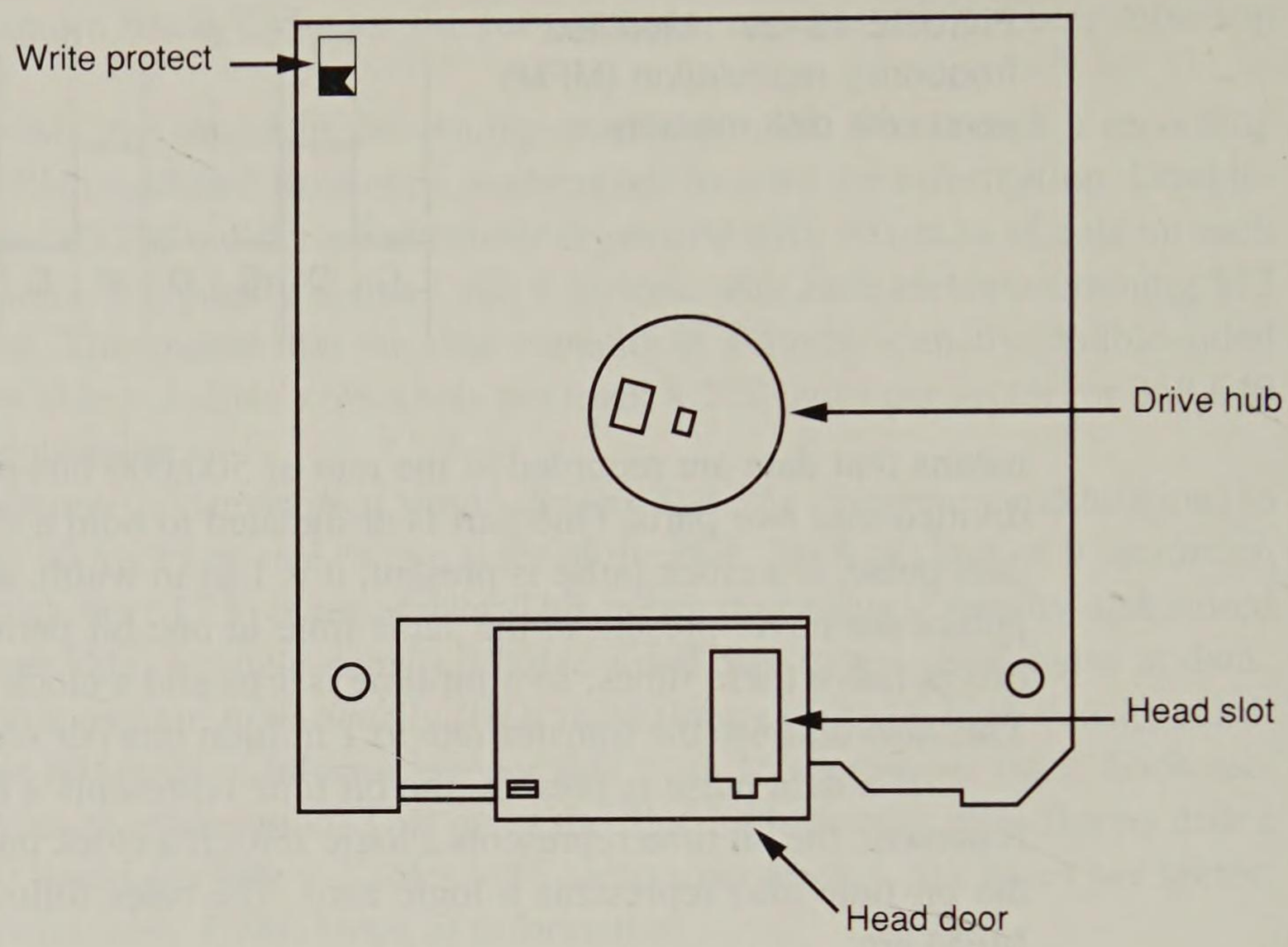



**FIGURE 13–30** The 3½″ micro-floppy disk.

tape write protection mechanism. To write protect (prevent writing) the micro-floppy disk, the plastic slide is moved to *open* the hole through the disk jacket. This allows light to strike a sensor which inhibits writing.

Still another improvement is the replacement of the index hole with a different drive mechanism. The drive mechanism on the mini-floppy allowed the disk drive to grab the disk at any point. This required an index hole so the electronics could find the beginning of a track. The index hole was trouble spot, because it collected dirt and dust. The micro-floppy has a drive mechanism that is keyed so that it only fits inside the disk drive one way. An index hole is no longer required. Because of the sliding head mechanism, and the fact that no index hole exists, the micro-floppy disk has no place to catch dust or dirt.

Two types of micro-floppy disks are widely available: the double-sided, double-density (*DSDD*) and the high-density (*HD*). The double-sided, double-density micro-floppy disk has 80 tracks per side with each track containing 9 sectors. Each sector contains 512 bytes of information. This allows 80 tracks per side × 2 sides × 9 sectors × 512 bytes per sector or 737,280 (720K) bytes of data to be stored on a double-density, double-sided floppy disk.

The high-density, double-sided micro-floppy disk stores even more information. It too has 80 tracks per side, but the number of sectors is doubled to 18 per track. This format still uses 512 bytes per sector as did the double-density format. The total number of bytes on a high-density, double-sided micro-floppy disk is 80 tracks per side × 2 sides × 18 sectors per track × 512 bytes per sector or 1,474,560 (1.44M) bytes of information.

Recently, a new 3½″ floppy disk has been introduced, the EHD (extended high-density) floppy disk. This new format stores 2.88M bytes of data on a single floppy disk. Currently, this format is expensive and it will take time for EHD disks to become common. Also available is the floptical disk, which stores data magnetically using an optical tracking system. The floptical disk stores 21M bytes of data.

## Hard Disk Memory

Larger disk memory is available in the form of the hard disk drive. The hard disk drive is often called a *fixed disk*, because it is not removable like the floppy disk. A hard disk is often called a *rigid disk*. The term *Winchester drive* is also used to describe a hard disk drive. Hard disk memory has a much larger capacity than floppy disk memory. Hard disk memory is available in sizes exceeding 1G byte of data. Common, low-cost (less than $1 per megabyte) sizes are presently 420M bytes or 525M bytes.

There are several differences between the floppy disk and the hard disk memory. The hard disk memory uses a flying head to store and read data from the surface of the disk. A flying head, which is very small and light, does not touch the surface of the disk. It flies above the surface on a film of air that is carried with the surface of the disk as it spins. The hard disk typically spins at 3,000 to 5,200 RPM, which is more than 10 times faster than the floppy disk. This higher rotational speed allows the head to fly (just as an airplane flies) just over the top of the surface of the disk. This is an important feature, because there is no wear on the surface, like there is with the floppy disk.

Problems can arise because of flying heads. One problem is a head crash. If the power is abruptly interrupted or the hard disk drive is jarred, the head can crash onto the disk surface. This can damage the disk surface or the head. To help prevent crashes, some drive manufacturers have included auto-parking heads, which automatically park the head when power is interrupted. When the heads are parked, they are moved to a safe landing zone (unused track)—the inner-most track—before power is disconnected. The inner-most track is a safe landing area because it is the very last track filled by the disk drive. Parking is the responsibility of the operator in this type of disk drive.

Another difference between a floppy disk drive and a hard disk drive is the number of heads and disk surfaces. A floppy disk drive has two heads, one for the upper surface and one for the lower surface. The hard disk drive may have up to eight disk surfaces (four platters) with up to two heads per surface. Each time a new cylinder is obtained by moving the head assembly, 16 new tracks are available under the heads. Refer to Figure 13–31 which illustrates a hard disk system.

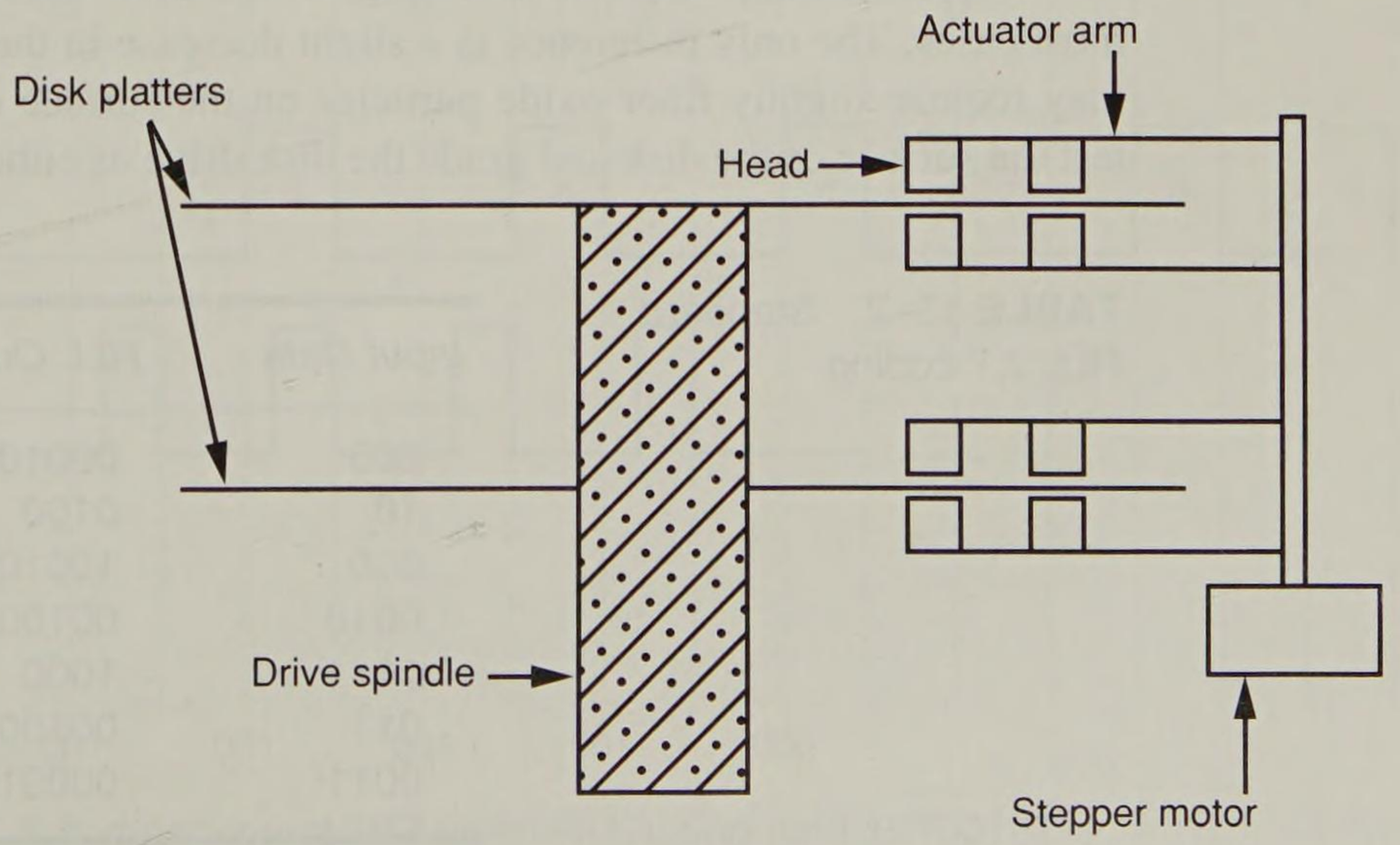



**FIGURE 13–31** A hard disk drive that uses four heads per platter.

Heads are moved from track to track using either a stepper motor or a voice coil. The stepper motor is slow and noisy, while the voice coil mechanism is quiet and quick. In a system that uses a stepper motor to position the heads, moving the head assembly requires one step per cylinder. In a system that uses a voice coil, the heads can be moved through many cylinders with one sweeping motion. This makes the disk drive faster when seeking new cylinders.

Another advantage of the voice coil system is that a servo mechanism can monitor the amplitude of the signal as it comes from the read head and make slight adjustments in the position of the heads. This is not possible with a stepper motor, which relies strictly on mechanics to position the head. Stepper motor type head positioning mechanisms can often become misaligned with use, while the voice coil mechanism corrects for any misalignment.

Hard disk drives often store information in sectors that are 1,024 bytes in length. Data are addressed in *clusters* of four sectors which contain 4,096 bytes on most hard disk drives. Hard disk drives use either MFM or RLL to store information. MFM is described with floppy disk drives. *Run-length limited* (RLL) is described here.

A typical MFM hard disk drive uses 18 sectors per track, so that 18K bytes of data are stored per track. If a hard disk drive has a capacity of 40M bytes it contains approximately 2,280 tracks. If the disk drive has two heads, this means it contains 1,140 cylinders. If it contains four heads, then it has 570 cylinders. These specifications vary from disk drive to disk drive.

*RLL Storage.* Run-length limited (RLL) disk drives use a different method for encoding the data than MFM. The term *RLL* means that the run of zeros (zeros in a row) is limited. A common RLL encoding scheme in use today is *RLL 2,7*. This means that the run of zeros is always between two and seven. Table 13–2 illustrates the coding used with standard RLL.

Data are first encoded using Table 13–2 before being sent to the drive electronics for storage on the disk surface. Because of this encoding technique, it is possible to achieve a 50 percent increase in data storage on a disk drive when compared to MFM. The main difference is that the RLL drive often contains 27 tracks instead of the 18 found on the MFM drive. Some RLL drives also use 35 sectors per track.

RLL encoding requires no change to the drive electronics or surface of the disk in most cases. The only difference is a slight decrease in the pulse width using RLL, which may require slightly finer oxide particles on the surface of the disk. Disk manufacturers test the surface of the disk and grade the disk drive as either an MFM-certified or an RLL-

**TABLE 13–2** Standard RLL 2,7 coding

| *Input Data* | *RLL Output* |
|---|---|
| 000 | 000100 |
| 10 | 0100 |
| 010 | 100100 |
| 0010 | 00100100 |
| 11 | 1000 |
| 011 | 001000 |
| 0011 | 00001000 |

certified drive. Other than grading, there is no difference in the construction of the disk drive or the magnetic material which coats the surface of the disks.

Figure 13–32 shows a comparison of MFM data and RLL data. Notice that the amount of time (space) required to store RLL data is less than the space required to store MFM data. Here a 101001011 is coded in both MFM and RLL, so that these two standards can be compared. Notice that the width of the RLL signal has been reduced so that 3 pulses fit in the same space as a clock and a data pulse for MFM. A 40M-byte MFM disk can hold 60M bytes of RLL encoded data. Besides holding more information, the RLL drive can be written and read at a higher rate.

All hard disk drives use either MFM or RLL encoding. There are a number of disk drive interfaces in use today. The oldest is the ST-506 interface, which uses either MFM or RLL data. A disk system using this interface is called *either MFM or RLL disk system.*

Newer standards are also in use today. These include ESDI, SCSI, and IDE. All of these newer standards use RLL, even though they normally do not call attention to it. The main difference is the interface between the computer and the disk drive.

The *enhanced small disk interface* (ESDI) system is capable of transferring data between itself and the computer at rates approaching 10M bytes per second. An ST-506 interface can approach a transfer rate of 860K bytes per second.

The *small computer system interface* (SCSI) system is also used because it allows up to seven different disk or other interfaces to be connected to the computer through the same interface controller. SCSI is found in some PC-type computers and also in the Apple Macintosh system. An improved version, SCSI-II, has started to appear in some systems.

The newest system is *integrated drive electronics* (IDE), which incorporates the disk controller in the disk drive and attaches the disk drive to the host system through a small interface cable. This allows many disk drives to be connected to a system without worrying about bus conflicts or controller conflicts. The IDE interface is also capable of driving other I/O devices besides the hard disk. This interface also usually contains at least a 32K-

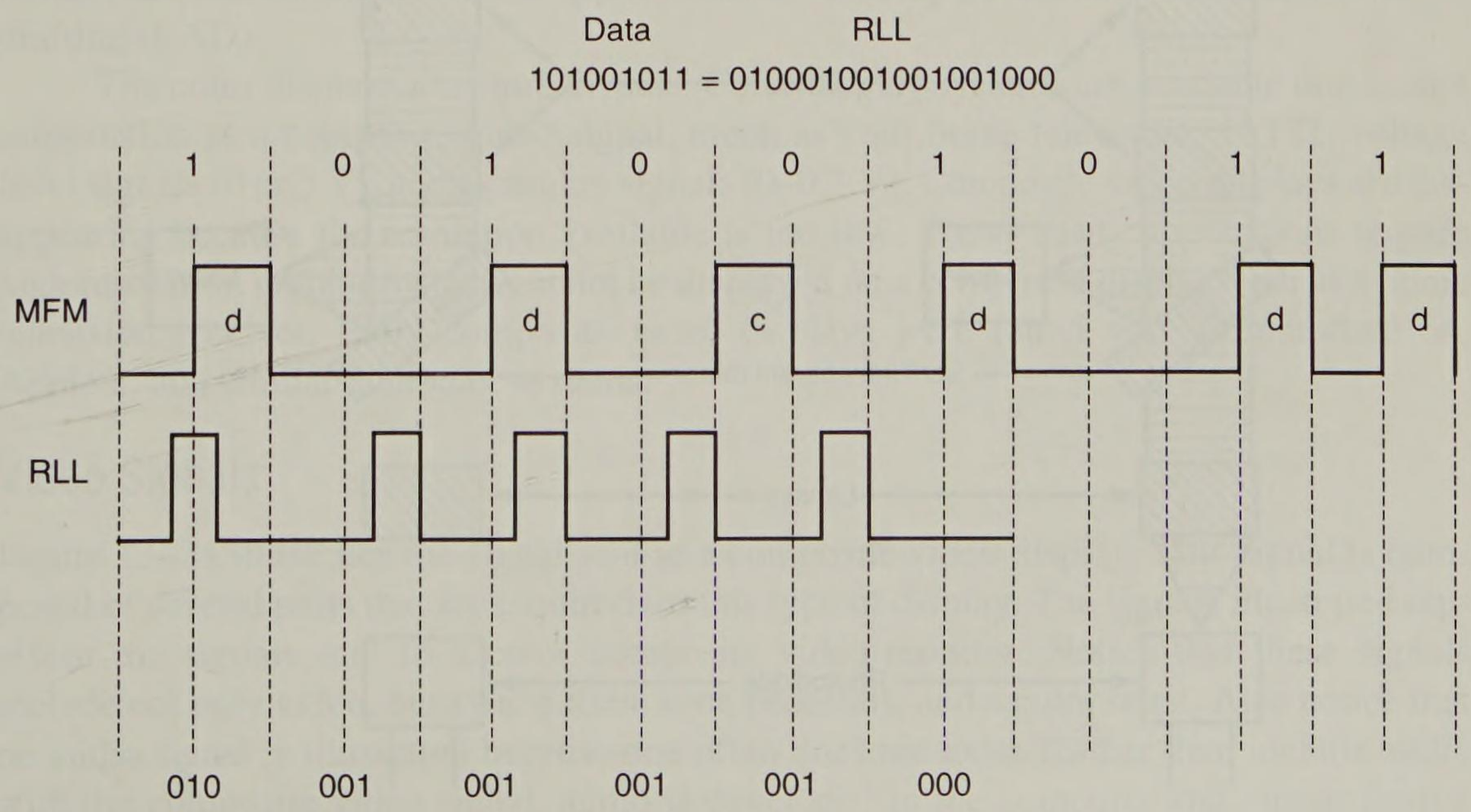



**FIGURE 13–32** A comparison of MFM with RLL using data 101001011.

byte cache memory for disk data. The cache speeds disk transfers. Common access times for an IDE drive are often less than 12 ms, where the access time for a floppy disk is about 200 ms. IDE drives are found in newer IBM Ps-2 systems and many clones, and they are becoming the standard hard disk memory interface.

## Optical Disk Memory

Optical disk memory (see Figure 13–33) is commonly available in two forms: the *CD-ROM* (compact disk/read only memory) and the *WORM* (write once/read mostly). The CD-ROM is the lowest-cost optical disk, but it suffers from lack of speed. Access times for a CD-ROM are typically 300 ms or longer, about the same as a floppy disk. (Slower CD-ROM devices are on the market and should be avoided.) Hard disk magnetic memory can have access times as little as 16 ms. The CD-ROM also suffers from lack of software applications at this time. The CD-ROM is available with large volume data storage such as the Bible, encyclopedias, clip art, magazine articles, etc. None of these applications have wide appeal at the current prices. At some future date, if access times decrease and more applications are introduced, the CD-ROM may become very popular. A CD-ROM stores 660M bytes of data or a combination of data and musical passages. As systems develop and become more visually active, the use of the CD-ROM drive will become more common.

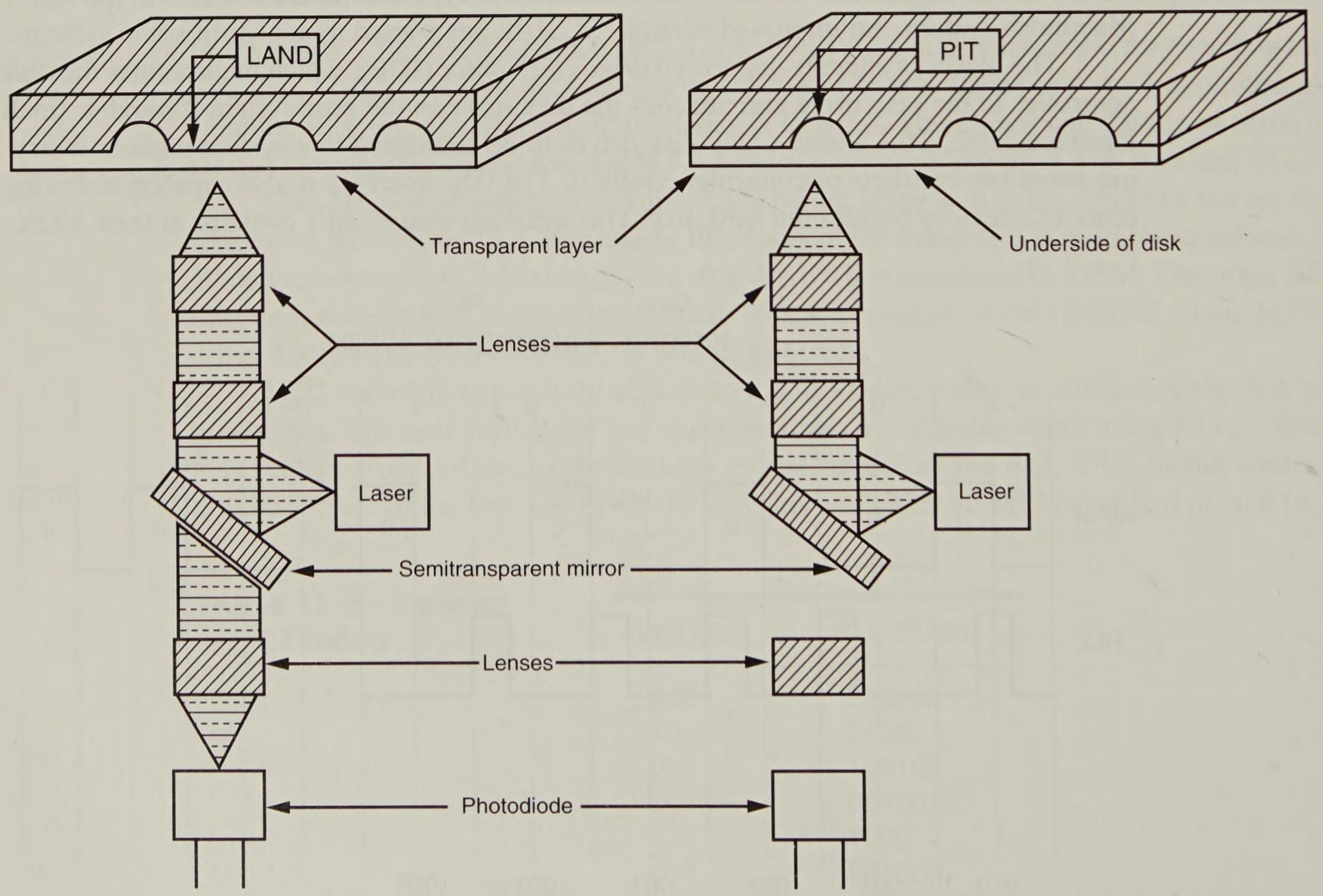



**FIGURE 13–33** The optical CD-ROM memory system.

The WORM drive has far more commercial application than the CD-ROM. The problem is that its application is very specialized due to the nature of the WORM. Because data may only be written once, its main application is in the banking industry, insurance industry, and other massive data storing organizations. The WORM is normally used to form an audit trail of transactions that are spooled onto the WORM and retrieved only during an audit. You might call the WORM an archiving device.

Many WORM and read/write optical disk memory systems are interfaced to the microprocessor using the SCSI or ESDI interface standards with hard disk memory. The difference is that the current optical disk drives are no faster than most floppy drives. Some CD-ROM drives are interfaced to the microprocessor through proprietary interfaces that are not compatible with other disk drives.

The main advantage of the optical disk is its durability. Because a solid state laser beam is used to read the data from the disk, and the focus point is below a protective plastic coating, the surface of the disk may contain small scratches and dirt particles and still be read correctly. This feature makes the optical disk easier to care for than a comparable floppy disk. About the only way to destroy data on an optical disk is to break the disk or deeply scar it.

## 13–5 VIDEO DISPLAYS

Modern video displays are *OEM* (original equipment manufacturer) devices that are usually purchased and incorporated into a system. Today there are many different types of video displays available, both color and monochrome.

Monochrome versions usually display information using amber, green, or paper-white displays. The paper-white display is becoming extremely popular for many applications. The most common of these applications are desktop publishing and computer-aided drafting (CAD).

The color displays are more diverse. Color display systems are available that accept information as a composite video signal, much as your home television, as TTL voltage level signals (0 or 5 V), and as analog signals (0–0.7 V). Composite video displays are disappearing because the resolution available is too low. Today many applications require high resolution graphics which cannot be displayed on a composite display such as a home television receiver. Early composite video displays were found with Commodore 64, Apple 2, and similar computer systems.

### Video Signals

Figure 13–34 illustrates the signal sent to a composite video display. This signal is composed of several parts that are required for this type of display. The signals illustrated represent the signals sent to a color composite video monitor. Notice that these signals include not only video, but sync pulses, sync pedestals, and a color burst. Also notice that no audio signal is illustrated because one often does not exist. Rather than include audio with the composite video signal, audio is developed in the computer and output from a speaker inside the computer cabinet. The major disadvantage of the composite video dis-

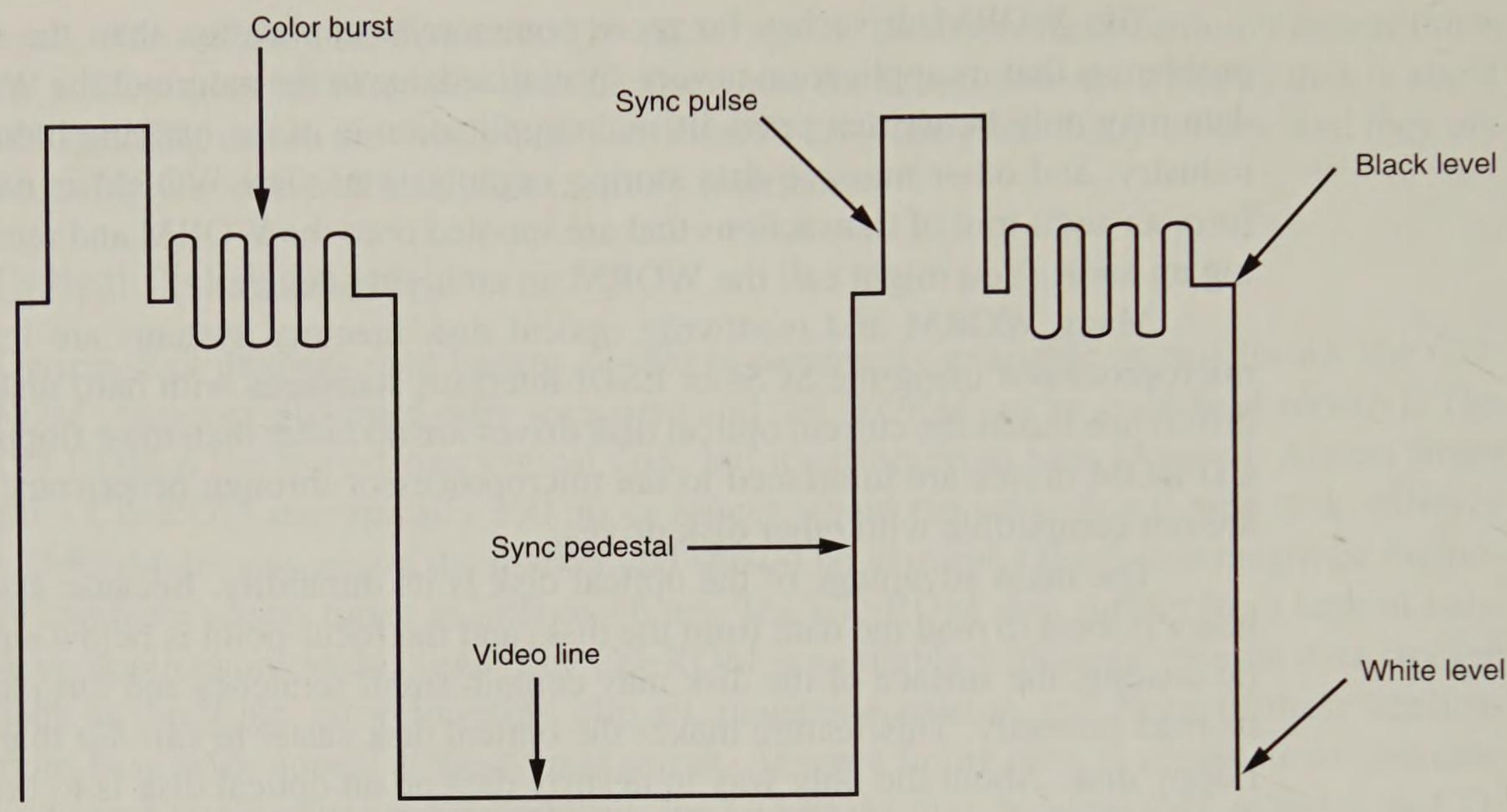



**FIGURE 13–34** The composite video signal.

play is the resolution and color limitations. Composite video signals were designed to emulate television video signals, so that a television receiver could function as a video monitor.

Most modern video systems use direct video signals generated with separate sync signals. In a direct video system, video information is passed to the monitor through a cable that uses separate lines for video and synchronization pulses. Recall that these signals were combined in a composite video signal.

A monochrome (one color) monitor uses one wire for video, one for horizontal sync, and one for vertical sync. Often these are the only signal wires found. A color video monitor uses three video signals. One signal represents red, another green, and the third blue. These monitors are often called RGB monitors for the video primary colors of light: red (R), green (G), and blue (B).

## The TTL RGB Monitor

The RGB monitor is available as either an analog or TTL monitor. The RGB monitor uses TTL level signals (0 or 5 V) as video inputs and a fourth line called intensity to allow a change in intensity. The RGB video TTL display can display a total of 16 different colors. The TTL RGB monitor is used in the *CGA* (color graphics adaptor) system found in older computer systems.

Table 13–3 lists these 16 colors and the TTL signals present to generate them. Eight of the 16 colors are generated at high intensity, and the other eight at low intensity. The three video colors are red, green, and blue. These are primary colors of light. The secondary colors are cyan, magenta and yellow. Cyan is a combination of blue and green video signals and is blue-green in color. Magenta is a combination of blue and red video signals and is a purple color. Yellow (high intensity) and brown (low intensity) are both a combination of red and green video signals. If additional colors are desired, TTL video is not normally

**TABLE 13–3** The 16 colors available on a TTL color (CGA) RGB display

| *Red* | *Green* | *Blue* | *Intensity* | *Color* |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Black |
| 0 | 0 | 0 | 1 | Gray |
| 0 | 0 | 1 | 0 | Light blue |
| 0 | 0 | 1 | 1 | Blue |
| 0 | 1 | 0 | 0 | Light green |
| 0 | 1 | 0 | 1 | Green |
| 0 | 1 | 1 | 0 | Light cyan |
| 0 | 1 | 1 | 1 | Cyan |
| 1 | 0 | 0 | 0 | Light red |
| 1 | 0 | 0 | 1 | Red |
| 1 | 0 | 1 | 0 | Light magenta |
| 1 | 0 | 1 | 1 | Magenta |
| 1 | 1 | 0 | 0 | Brown |
| 1 | 1 | 0 | 1 | Yellow |
| 1 | 1 | 1 | 0 | White |
| 1 | 1 | 1 | 1 | Intense white |

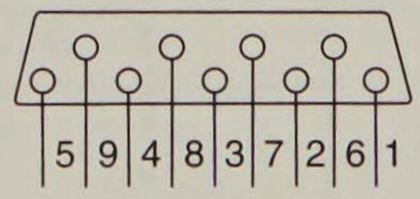
used. A scheme that provided 32 colors was developed using low and medium color TTL video signals, but it proved of little application and never found widespread use.

Figure 13–35 illustrates the connector most often found on a TTL RGB monitor or a TTL monochrome monitor. The connector illustrated is a 9-pin connector. Two of the connections are used for ground, three for video, two for synchronization or retrace signals, and one for intensity. Notice that pin 7 is labeled normal video. This is the pin used on a monochrome monitor for the luminance, or brightness, signal. Monochrome TTL monitors use the same 9-pin connector as RGB TTL monitors.

## The Analog RGB Monitor

An analog video display is required to display more than 16 colors. These are often called analog RGB monitors. Analog RGB monitors have three video input signals, but don't

**FIGURE 13–35** The 9-pin connector found on a TTL monitor.




| Pin | Function |
|---|---|
| 1 | Ground |
| 2 | Ground |
| 3 | Red video |
| 4 | Green video |
| 5 | Blue video |
| 6 | Intensity |
| 7 | Normal video |
| 8 | Horizontal retrace |
| 9 | Vertical retrace |

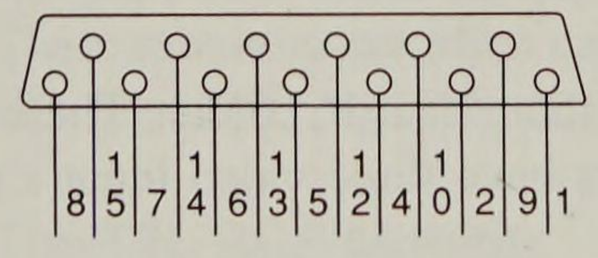
have the intensity input. Because the video signals are analog signals instead of two-level TTL signals, they are any voltage level between 0.0 V and 0.7 V. This allows an infinite number of colors to be displayed. This is because an infinite number of voltage levels between the minimum and maximum could be generated. In practice, a finite number of levels are generated. This is usually either 256K, 16M, or 24M colors, depending on the standard.

Figure 13–36 illustrates the connector used for an analog RGB or analog monochrome monitor. Notice that the connector has 15 pins and supports both RGB and monochrome analog displays. The way data are displayed on an analog RGB monitor depends on the interface standard used with the monitor. Pin 9 is a key, which means that no hole exists on the female connector for this pin.

Most analog displays use a *digital-to-analog converter* (DAC) to generate each color video voltage. A common standard uses a 6-bit DAC for each video signal to generate 64 different voltage levels between 0 V and 0.7 V. There are 64 different red video levels, 64 different green video levels, and 64 different blue video levels. This allows $64 \times 64 \times 64$, or 262,144 (256K) colors to be displayed.

Other arrangements are possible, but the speed of the DAC is critical. Most modern displays require an operating conversion time of 25 ns to 40 ns maximum. When converter technology advances, additional resolution at a reasonable price will become available. If 7-bit converters are used for generating video, $128 \times 128 \times 128$, or 2,097,152 (2M) colors are displayed. In this system a 21-bit color code is needed, so that a 7-bit code is applied to each DAC. Eight-bit converters also find applications and allow $256 \times 256 \times 256$, or 16,777,216 (16M) colors.

Figure 13–37 illustrates the video generation circuit employed in many common video standards such as *EGA* (enhanced graphics adaptor) and *VGA* (variable graphics array) as used with an IBM-PC. This circuit is used to generate VGA video. Notice that




| Pin | Function |
|---|---|
| 1 | Red video |
| 2 | Green video (monochrome video) |
| 3 | Blue video |
| 4 | Ground |
| 5 | Ground |
| 6 | Red ground |
| 7 | Green ground (monochrome ground) |
| 8 | Blue ground |
| 9 | Blocked as a key |
| 10 | Ground |
| 11 | Color detect (ground on a color monitor) |
| 12 | Monochrome detect (ground on a monochrome monitor) |
| 13 | Horizontal retrace |
| 14 | Vertical retrace |
| 15 | Ground |

**FIGURE 13–36** The 15-pin connector found on an analog monitor.

each color is generated with an 18-bit digital code. Six of the 18 bits are used to generate each video color voltage when applied to the inputs of a 6-bit DAC.

A high-speed palette SRAM (access time of less than 40 ns) is used to store 256 different 18-bit codes that represent 256 different hues. The 18-bit codes are applied to the digital-to-analog converters. The address input to the SRAM selects one of the 256 colors

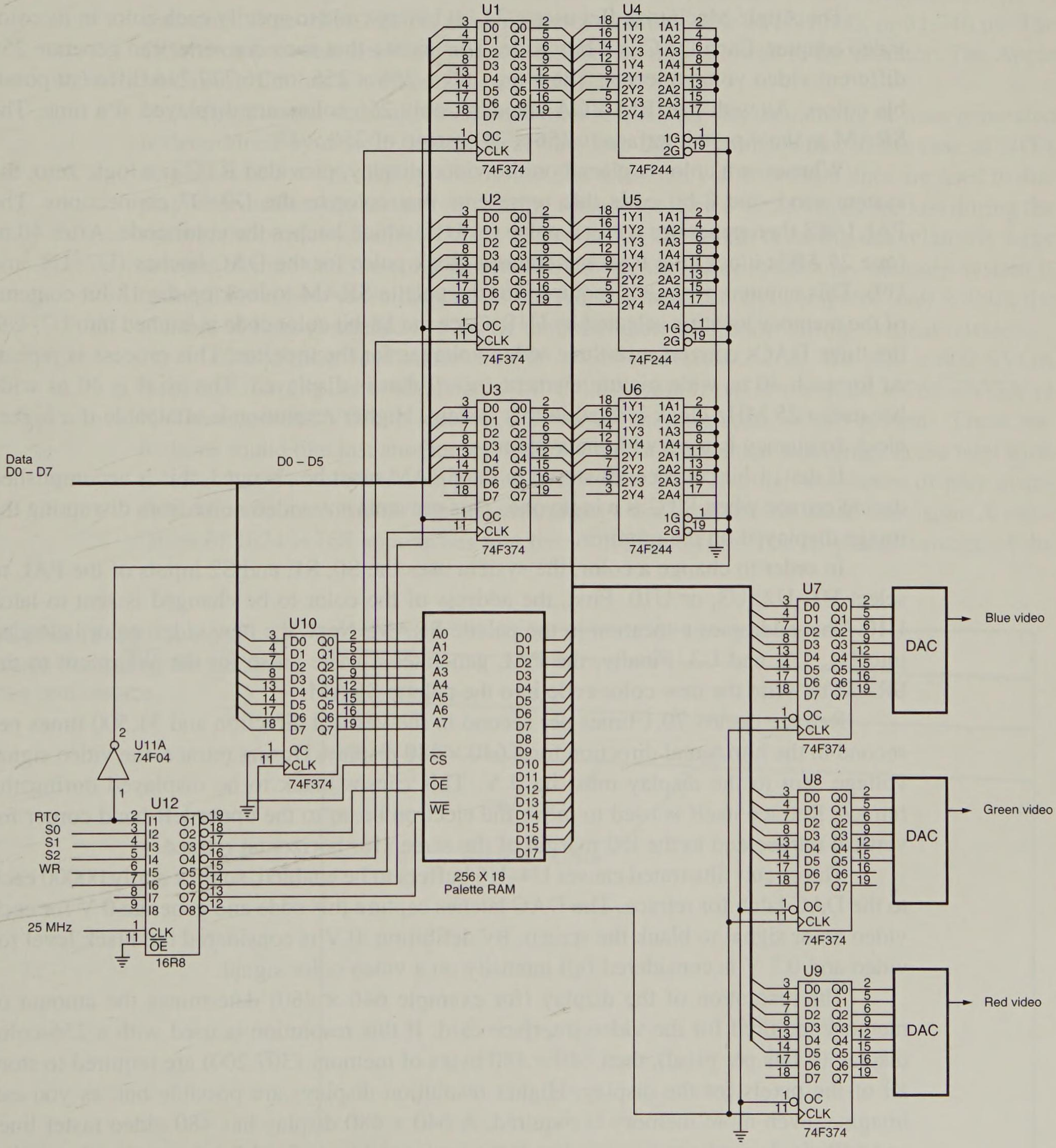



**FIGURE 13–37** Generation of VGA video signals.

stored as 18-bit binary codes. This system allows 256 colors, out of a possible 256K colors, to be displayed at one time. In order to select any of 256 colors, an 8-bit code stored in the computer's video display RAM is used to specify the color of a picture element. If more colors are used in a system, the code must be wider. For example, a system that displays 1,024 colors requires a 10-bit code to address the SRAM, which contains 1,024 locations, each containing an 18-bit color code. Some newer systems use a larger-palette SRAM, to store up to 64K different color codes.

The Apple Macintosh IIci uses a 24-bit binary code to specify each color in its color video adapter. Each DAC is 8 bits wide. This means that each converter can generate 256 different video voltage levels. There are $256 \times 256 \times 256$, or 16,777,216 different possible colors. As with the IBM VGA standard, only 256 colors are displayed at a time. The SRAM in the Apple interface is $256 \times 24$ instead of $256 \times 18$.

Whenever a color is placed on the video display, provided RTC is a logic zero, the system sends the 8-bit code that represents that color to the D0–D7 connections. The PAL16R8 then generates a clock pulse for U10 which latches the color code. After 40 ns (one 25 MHz clock) the PAL generates a clock pulse for the DAC latches (U7, U8, and U9). This amount of time is required for the palette SRAM to look up the 18-bit contents of the memory location selected by U10. Once the 18-bit color code is latched into U7–U9, the three DACs convert it to three video voltages for the monitor. This process is repeated for each 40 ns wide picture element (*pixel*) that is displayed. The pixel is 40 ns wide because a 25 MHz clock is used in this system. Higher resolution is attainable if a higher clock frequency is used with the system.

If the 18-bit color codes stored in the SRAM must be changed, this is accomplished during retrace when RTC is a logic one. This prevents any video noise from disrupting the image displayed on the monitor.

In order to change a color, the system uses the S0, S1, and S2 inputs of the PAL to select U1, U2, U3, or U10. First, the address of the color to be changed is sent to latch U10. This addresses a location in the palette SRAM. Next, the new video color is loaded into U1, U2, and U3. Finally, the PAL generates a write pulse for the $\overline{WE}$ input to the SRAM to write the new color code into the palette SRAM.

Retrace occurs 70.1 times per second in the vertical direction and 31,500 times per second in the horizontal direction for a $640 \times 480$ display. During retrace, the video signal voltage sent to the display must be 0 V. This causes black to be displayed during the retrace. Retrace itself is used to move the electron beam to the upper left-hand corner for vertical retrace and to the left margin of the screen for horizontal retrace.

The circuit illustrated causes U4–U6 buffers to be enabled, so they apply 00000 each to the DAC latch for retrace. The DAC latches capture this code and generate 0 V for each video color signal to blank the screen. By definition, 0 V is considered the black level for video and 0.7 V is considered full intensity on a video color signal.

The resolution of the display (for example $640 \times 480$) determines the amount of memory required for the video interface card. If this resolution is used with a 256-color display (8 bits per pixel), then $640 \times 480$ bytes of memory (307,200) are required to store all of the pixels for the display. Higher resolution displays are possible but, as you can imagine, even more memory is required. A $640 \times 480$ display has 480 video raster lines and 640 pixels per line. A *raster line* is the horizontal line of video information that is displayed on the monitor. A pixel is the smallest subdivision of this horizontal line.

Figure 13–38 illustrates the video display showing the video lines and retrace. The slant of each video line in this illustration is greatly exaggerated, as is the spacing between lines. This illustration shows retrace in both vertical and horizontal directions. In the case of a VGA display, the vertical retrace occurs exactly 70.1 times per second and the horizontal retrace occurs exactly 31,500 times per second. (The Apple Macintosh IIci uses a vertical rate of 66.67 Hz and a horizontal rate of 35 KHz to generate a 640 × 480 color display.)

In order to generate 640 pixels across one line, it takes 40 ns × 640, or 25.6 µs. A horizontal time of 31,500 Hz allows a horizontal line time of 1/31,500, or 31.746 µs. The difference between these two times is the retrace time allowed to the monitor. The Apple Macintosh IIci has a horizontal line time of 28.57 µs.

Because the vertical retrace repetition rate is 70.1 Hz, the number of lines generated is determined by dividing the vertical time into the horizontal time. In the case of VGA display, a 640 × 400 display, this is 449.358 lines. Only 400 of these lines are used to display information; the rest are lost during the retrace. Since 49.358 lines are lost during the retrace, the retrace time is 49.358 × 31.766 µs, or 1568 µs. It is during this relatively large amount of time that the color palette SRAM is changed or the display memory system is updated for a new video display. In the Apple Macintosh IIci computer (640 × 480), the number of lines generated is 525. Of the total lines, 45 are lost during vertical retrace.

Other display resolutions are 800 × 600 and 1024 × 768. The 800 × 600 SVGA (super VGA) display is ideal for a 14″ color monitor, while the 1024 × 768 EVGA or XVGA (extended VGA) is ideal for a 21″ or 25″ monitor used in CAD systems. These resolutions sound like just another set of numbers, but realize that an average home television receiver has a resolution of approximately 400 × 300. The high-resolution display available on computer systems is much clearer than that available on home television. A resolution of 1024 × 768 approaches that found in 35mm film. The only disadvantage of the

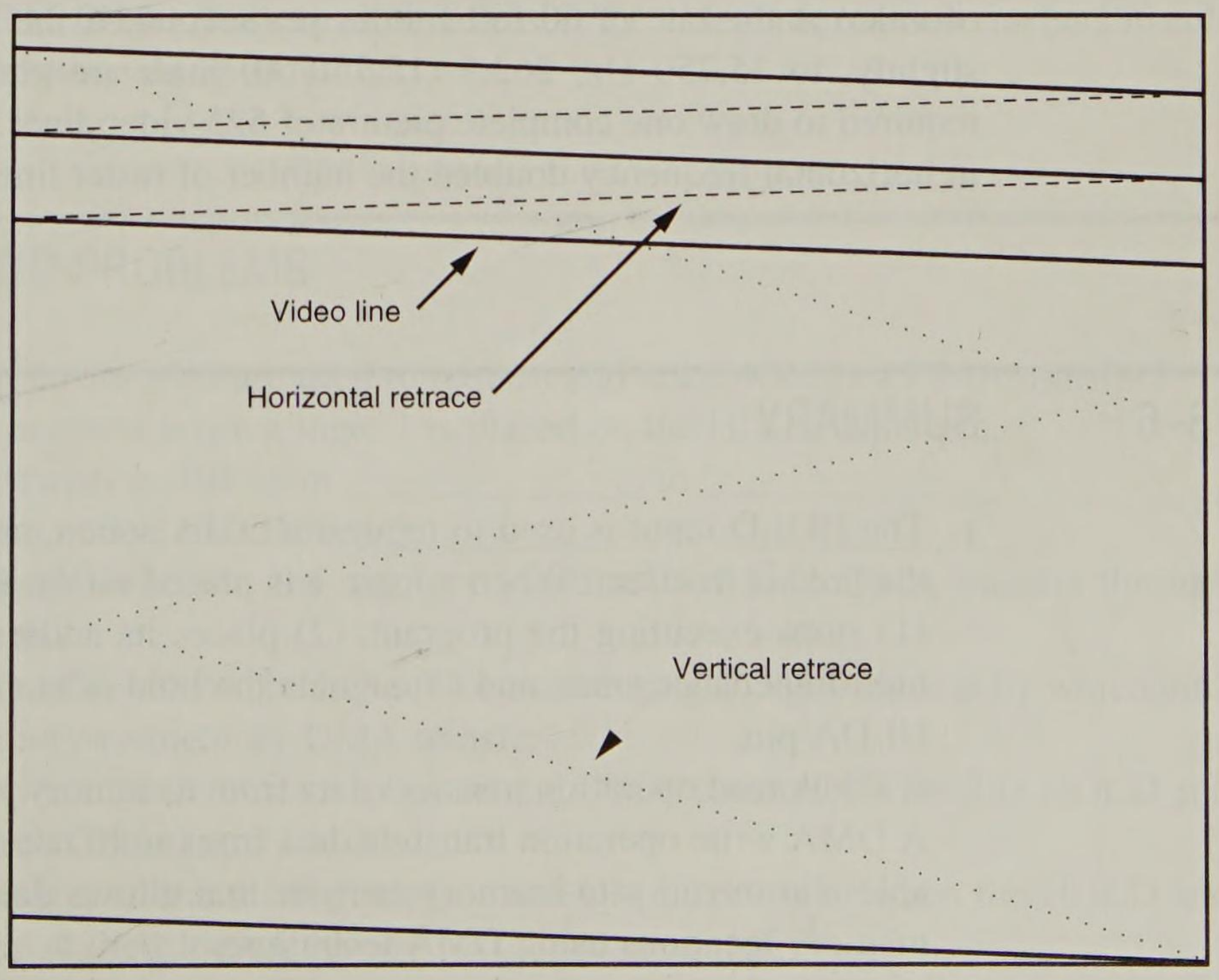



**FIGURE 13–38** A video screen illustrating the raster lines and retrace.

video display on a computer screen is the number of colors displayed at a time, but as time passes, this will surely improve. Additional colors allow the image to appear more realistic because of subtle shadings that are required for a high-quality lifelike image.

If a display system operates with a 60 Hz vertical time and a 15,600 Hz horizontal time, the number of lines generated is 15,600/60, or 260 lines. The number of usable lines in this system is most likely 240, where 20 are lost during vertical retrace. It is clear that the number of scanning lines is adjustable by changing the vertical and horizontal scanning rates.

The vertical scanning rate must be greater than or equal to 50 Hz, or flickering will occur. The vertical rate must not be higher than about 75 Hz, or problems with the vertical deflection coil may occur. The electron beam in a monitor is positioned by an electrical magnetic field generated by coils in a yoke that surrounds the neck of the picture tube. Since the magnetic field is generated by coils, the frequency of the signal applied to the coil is limited.

The horizontal scanning rate is also limited by the physical design of the coils in the yoke. Because of this, it is normal to find a narrow range of frequencies applied to the horizontal coils—usually 30,000 Hz–37,000 Hz or 15,000 Hz–17,000 Hz. Some newer monitors are called multi-sync monitors because the deflection coil is taped, so that it can be driven with different deflection frequencies. Sometimes both the vertical and horizontal coils are taped for different vertical and horizontal scanning rates.

High-resolution displays use either interlaced or non-interlaced scanning. The non-interlaced scanning system is used in all standards except the highest. In the interlaced system, the video image is displayed by drawing half the image first, with all of the odd scanning lines, then drawing the other half, using the even scanning lines. Obviously this system is more complex, and is more efficient because the scanning frequencies are reduced by 50 percent. For example, a video system that uses 60 Hz for the vertical scanning frequency and 15,720 Hz for the horizontal frequency generates 262 (15,720/60) lines of video at the rate of 60 full frames per second. If the horizontal frequency is changed slightly, to 15,750 Hz, 262.5 (15,750/60) lines are generated, so two full sweeps are required to draw one complete picture of 525 video lines. Notice how just a slight change in horizontal frequency doubled the number of raster lines.

## 13–6 SUMMARY

1. The HOLD input is used to request a DMA action, and the HLDA output signals that the hold is in effect. When a logic 1 is placed on the HOLD input, the microprocessor (1) stops executing the program, (2) places its address, data, and control bus at their high-impedance state, and (3) signals the hold is in effect by placing a logic 1 on the HLDA pin.
2. A DMA read operation transfers data from a memory location to an external I/O device. A DMA write operation transfers data from an I/O device into the memory. Also available is a memory-to-memory transfer that allows data to be transferred between two memory locations using DMA techniques.

3. The 8237 direct memory access (DMA) controller is a four-channel device that can be expanded to include additional channels of DMA.
4. Disk memory comes in the form of floppy disk storage, which is found as either the 5¼″ mini-floppy disk or the 3½″ micro-floppy disk. Both disks can be found double-sided, double density (DSDD) or high-density (HD). The DSDD 5¼″ disk stores 360K bytes of data, and the HD 5¼″ disk stores 1.2M bytes of data. The DSDD 3½″ disk stores 720K bytes of data, and the HD 3½″ disk stores 1.44M bytes of data.
5. Floppy disk memory data are stored using NRZ (nonreturn to zero) recording. This method saturates the disk with one polarity of magnetic energy for a logic one and the opposite polarity for a logic zero. In either case, the magnetic field never returns to zero. This technique eliminates the need for a separate erase head.
6. Data are recorded on disks by using either modified frequency modulation (MFM) or run-length limited (RLL) encoding schemes. The MFM encoding scheme records a data pulse for a logic one, no data or clock for the first logic zero of a string of zeros, and a clock pulse for the second and subsequent logic zeros in a string of zeros. The RLL encoding scheme encodes data so 50 percent more information can be packed onto the same disk area. Most modern disk memory systems use the RLL encoding scheme.
7. Video monitors are either TTL or analog. The TTL monitor uses two discrete voltage levels of 0 V and 5.0 V. The analog monitor uses an infinite number of voltage levels between 0.0 V and 0.7 V. An analog monitor can display an infinite number of video levels, while a TTL monitor is limited to two video levels.
8. The color TTL monitor displays 16 different colors. This is accomplished through three video signals (red, green, and blue) and an intensity input. The analog color monitor can display an infinite number of colors through its three video inputs. In practice, the most common form of color analog display system, VGA, can display 256K different colors.
9. The video standards found today include VGA (640 × 480), SVGA (800 × 600), and EVGA or XVGA (1024 × 768). In all three cases, the video information can be 256 colors out of a total possible 256K colors.

## 13–7 QUESTIONS AND PROBLEMS

1. What microprocessor pins are used to request and acknowledge a DMA transfer?
2. Explain what happens when a logic 1 is placed on the HOLD input pin.
3. A DMA read transfers data from _____ to _____.
4. A DMA write transfers data from _____ to _____.
5. The DMA controller selects the memory location used for a DMA transfer through what bus signals?
6. The DMA controller selects the I/O device used during a DMA transfer by which pin?
7. What is a memory-to-memory DMA transfer?
8. Describe the effect on the microprocessor and DMA controller when the HOLD and HLDA pins are at their logic 1 levels.
9. Describe the effect on the microprocessor and DMA controller when the HOLD and HLDA pins are at their logic 0 levels.

10. The 8237 DMA controller is a _____ channel DMA controller.
11. If the 8237 DMA controller is decoded at I/O ports 2000H–200FH, what ports are used to program channel 1?
12. Which 8237 DMA controller register is programmed to initialize the controller?
13. How many bytes can be transferred by the 8237 DMA controller?
14. Write a sequence of instructions that transfer data from memory location 21000H–210FFH to 20000H–200FFH, using channel 2 of the 8237 DMA controller. You must initialize the 8237 and use the latch described in Section 13–1 to hold A19–A16.
15. Write a sequence of instructions that transfer data from memory to an external I/O device, using channel 3 of the 8237. The memory area to be transferred is at location 20000H–20FFFH.
16. The $5\frac{1}{4}''$ disk is known as a _____-floppy disk.
17. The $3\frac{1}{2}''$ disk is known as a _____-floppy disk.
18. Data are recorded in concentric rings on the surface of a disk known as _____.
19. A track is divided into sections of data called _____.
20. On a double-sided disk, the upper and lower tracks together are called a _____.
21. Why is NRZ recording used on a disk memory system?
22. Draw the timing diagram generated to write a 1001010000, using MFM encoding.
23. Draw the timing diagram generated to write a 1001010000, using RLL encoding.
24. What is a flying head?
25. Why must the heads on a hard disk be parked?
26. What is the difference between a voice-coil head-positioning mechanism and a stepper-motor head-positioning mechanism?
27. What is a WORM?
28. What is a CD-ROM?
29. What is the difference between a TTL monitor and an analog monitor?
30. What are the three primary colors of light?
31. What are the three secondary colors of light?
32. What is a pixel?
33. A video display with a resolution of 800 × 600 contains _____ lines of video information, with each line divided into _____ pixels.
34. Explain how a TTL RGB monitor can display 16 different colors.
35. Explain how an analog RGB monitor can display an infinite number of colors.
36. If an analog RGB video system uses 7-bit DACs, it can generate _____ different colors.
37. Why does standard VGA allow only 256 different colors out of 256K colors to be displayed at one time?
38. If a video system uses a vertical frequency of 60 Hz and a horizontal frequency of 32,400 Hz, how many raster lines are generated?

# CHAPTER 14

# The Pentium Microprocessor

## INTRODUCTION

The Pentium microprocessor signals an improvement to the architecture found in the 80486 microprocessor. The changes include an improved cache structure, a wider data bus width, a faster numeric coprocessor, a dual integer processor, and branch prediction logic. The cache has been reorganized to form two caches that are each 8K bytes in size, one for caching data, the other for instructions. The data bus width has been increased from 32 bits to 64 bits. The numeric coprocessor operates about five times faster than the 80486 numeric coprocessor. The dual integer processor often allows 2 instructions per clock. Finally, the branch prediction logic allows programs that branch to execute more efficiently. Notice that these changes are internal to the Pentium, which makes software from earlier Intel 80X86 microprocessors upward compatible.

## OBJECTIVES

Upon completion of this chapter, you will be able to:

1. Contrast the Pentium with the 80386 and 80486 microprocessors.
2. Describe the organization and interface of the 64-bit wide Pentium memory system and its variations.
3. Contrast the changes in the memory management unit and paging unit when compared to the 80386 and 80486 microprocessors.
4. Detail the new instructions found with the Pentium microprocessor.
5. Explain how the superscaler dual integers unit improves performance of the Pentium microprocessor.
6. Describe the operation of the branch prediction logic.

## 14–1 INTRODUCTION TO THE PENTIUM MICROPROCESSOR

Before this or any other microprocessor can be used in a system, the function of each pin must be understood. This section of the chapter details the operation of each pin, along with the external memory system and I/O structures of the Pentium microprocessor.

Figure 14–1 illustrates the pinout of the Pentium microprocessor, which is packaged in a huge 237-pin PGA (pin grid array). Currently, the Pentium is available in a single version, but shortly the P24T version will become available. The P24T version will contain a 32-bit data bus compatible for insertion into 80486 machines that contain the P24T socket. The most notable difference in the pinout is that there are 64 data bus connections, which require a larger physical footprint.

As with earlier versions of the Intel family of microprocessors, the Pentium requires a single +5.0 V power supply for operation. The power supply current averages 3.3 A for the 66 MHz version of the Pentium, and 2.91 A for the 60 MHz version. Because these currents are significant, so are the power dissipations of these microprocessors: 13 W for the 66 MHz version and 11.9 W for the 60 MHz version. These currents will be reduced in the future with the release of a 3.3 V version and another version that will be constructed using smaller internal component sizes. These new versions operate at 90 MHz and 100 MHz. At present, a good heat sink with considerable airflow is required to keep the Pentium cool. The Pentium contains multiple Vcc and Vss connections that must all be connected to +5.0 V and ground for proper operation. Some of the pins are labeled N/C (no connection) and must not be connected.

Each Pentium output pin is capable of providing 4.0 mA of current at a logic 0 level, and 2.0 mA of current at a logic 1 level. This represents an increase in drive current compared to the 2.0 mA of current available on earlier 8086, 8088, and 80286 output pins. Each input pin represents a small load requiring only ±15 μA of current. In all but the smallest systems, these current levels require bus buffers.

The function of each Pentium group of pins follows:

$\overline{A20}$ (Address A20 Mask)—An input asserted in the real mode to signal the Pentium to perform address wraparound, as in the 8086 microprocessor for use of the HIMEM.SYS driver.

A31–A3 (Address Bus Connections)—Address any of the 512K × 64 memory locations found in the Pentium memory system. Note that A0, A1, and A2 are encoded in the bus enable ($\overline{BE7}$–$\overline{BE0}$), described elsewhere, to select any or all of the eight bytes in a 64-bit-wide memory location.

$\overline{ADS}$ (Address Data Strobe)—Becomes active whenever the Pentium has issued a valid memory or I/O address. This signal is combined with the W/$\overline{R}$ and M/$\overline{IO}$ signals to generate the separate read and write signals present in the earlier 8086–80286 microprocessor-based systems.

AHOLD (Address Hold)—An input that causes the Pentium to hold the address and AP for the next clock.

AP (Address Parity)—Provides even parity for the memory address on all Pentium-initiated memory and I/O transfers. The AP pin must also be

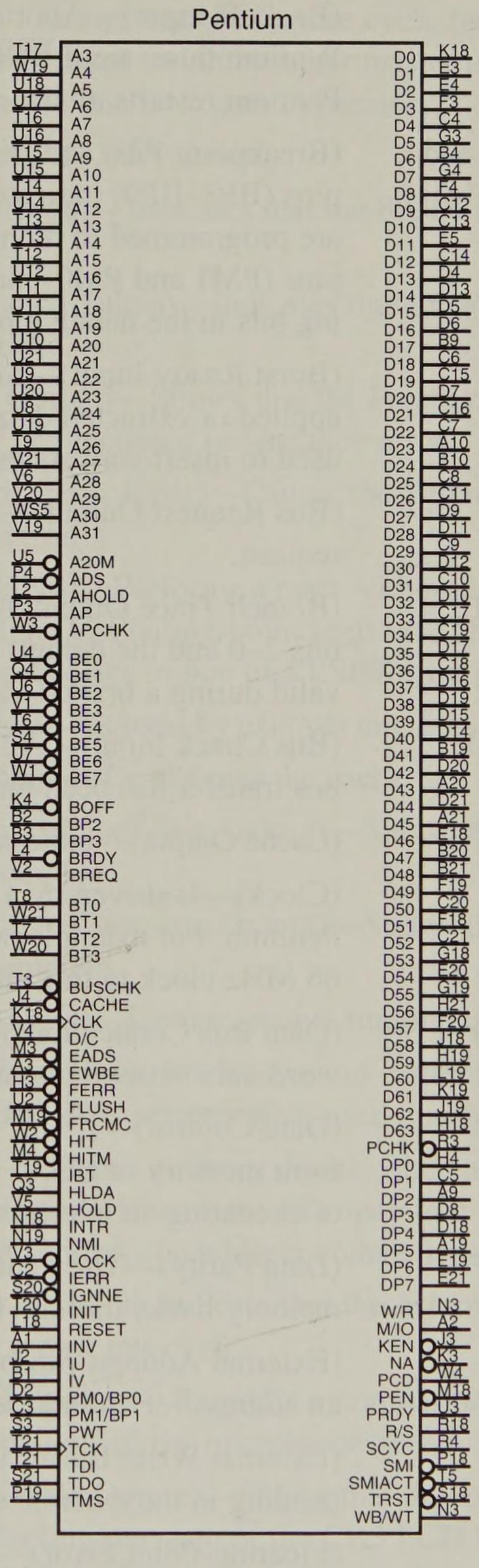



**FIGURE 14–1** The pinout of the Pentium microprocessor.

driven with even parity information on all inquire cycles, in the same clocking period as the $\overline{\text{EADS}}$ signal.

$\overline{\text{APCHK}}$ (Address Parity Check)—Becomes a logic one whenever the Pentium detects an address parity error.

$\overline{\text{BE7}}$–$\overline{\text{BE0}}$ (Bank Enable Signals)—Select the access of a byte, word, doubleword, or quadword of data. These signals are generated internally by the microprocessor from address bits A0, A1, and A2.

| | |
|---|---|
| $\overline{\text{BOFF}}$ | (Backoff Input)—Aborts all outstanding bus cycles and floats the Pentium buses until $\overline{\text{BOFF}}$ is negated. After $\overline{\text{BOFF}}$ is negated, the Pentium restarts all aborted bus cycles in their entirety. |
| BP[3:2] and PM/BP[1:0] | (Breakpoint Pins and Performance Monitoring Pins)—The breakpoint pins (BP3–BP0) indicate a breakpoint match when the debug registers are programmed to monitor for matches. The performance monitoring pins (PM1 and PM0) indicate the settings of the performance monitoring bits in the debug mode control register. |
| $\overline{\text{BRDY}}$ | (Burst Ready Input)—Signals the Pentium that the external system has applied or extracted data from the data bus connections. This signal is used to insert wait states into the Pentium timing. |
| BREQ | (Bus Request Output)—Indicates that the Pentium has generated a bus request. |
| BT3–BT0 | (Branch Trace Outputs)—Provide the branch target linear address on bits 2–0 and the default operand size on BT3. These outputs become valid during a branch trace special message cycle. |
| $\overline{\text{BUSCHK}}$ | (Bus Check Input)—Allows the system to signal the Pentium that the bus transfer has been unsuccessful. |
| $\overline{\text{CACHE}}$ | (Cache Output)—Indicates that the current Pentium cycle can cache data. |
| CLK | (Clock)—Is driven by a clock signal at the operating frequency of the Pentium. For example, to operate the Pentium at 66 MHz, we apply a 66 MHz clock to this pin. |
| D63–D0 | (Data Bus Connections)—Transfer byte, word, doubleword, and quadword data between the microprocessor and its memory and I/O system. |
| D/$\overline{\text{C}}$ | (Data/$\overline{\text{Control}}$)—If D/$\overline{\text{C}}$ is a logic 1, the data bus contains data for or from memory or I/O. If D/$\overline{\text{C}}$ is a logic 0, the microprocessor is halted or executing an interrupt acknowledge. |
| DP7–DP0 | (Data Parity)—-The Pentium generates and accepts parity for its eight memory banks through these connections. |
| $\overline{\text{EADS}}$ | (External Address Strobe Input)—Signals that the address bus contains an address for an inquire cycle. |
| $\overline{\text{EWBE}}$ | (External Write Buffer Empty Input)—Indicates that a write cycle is pending in the external system. |
| $\overline{\text{FERR}}$ | (Floating-Point Error)—Is compatible to the $\overline{\text{ERROR}}$ line in the 80386, to show that the internal coprocessor has erred. |
| $\overline{\text{FLUSH}}$ | (Flush Cache Input)—Causes the cache to flush all write-back lines and invalidate its internal caches. If the $\overline{\text{FLUSH}}$ input is a logic zero during a reset operation, the Pentium enters its test mode. |
| $\overline{\text{FRCMC}}$ | (Functional Redundancy Check)—Used to poll the Pentium to determine whether it is configured in the master or checker mode. |
| $\overline{\text{HIT}}$ | (Hit)—Shows that the internal cache contains valid data in the inquire mode. |

| | |
|---|---|
| $\overline{\text{HITM}}$ | (Hit Modified)—Shows that the inquire cycle found a modified cache line. This output is used to inhibit other master units from accessing data until the cache line is written to memory. |
| HOLD | (Hold)—Requests a DMA action. |
| HLDA | (Hold Acknowledge)—Indicates that the 80386 is currently in a hold condition. |
| IBT | (Instruction Branch Taken)—Indicates that the Pentium has taken an instruction branch. |
| $\overline{\text{IERR}}$ | (Internal Error Output)—Shows that the Pentium has detected an internal parity error or functional redundancy error. |
| $\overline{\text{IGNNE}}$ | (Ignore Numeric Error Input)—Causes the Pentium to ignore a numeric coprocessor error. |
| INIT | (Initialization Input)—Performs a reset without initializing the caches, write-back buffers, or floating-point registers. This may not be used to reset the microprocessor in lieu of RESET after power-up. |
| INTR | (Interrupt Request)—Is used by external circuitry to request an interrupt. |
| INV | (Invalidation Input)—Determines the cache line state after an inquiry. |
| IU | (U-Pipe Instruction Complete Output)—Shows that the instruction in the U-pipe is complete. |
| IV | (V-Pipe Instruction Complete Output)—Shows that the instruction in the V-pipe is complete. |
| $\overline{\text{KEN}}$ | (Cache Enable Input)—Used to enable internal caching. |
| $\overline{\text{LOCK}}$ | (Lock)—Becomes a logic 0 whenever an instruction is prefixed with the LOCK: prefix. This is most often used during DMA accesses. |
| M/$\overline{\text{IO}}$ | (Memory/IO)—Selects a memory device when a logic 1, or an I/O device when a logic 0. During the I/O operation the address bus contains a 16-bit I/O address on address connections A15–A3. |
| $\overline{\text{NA}}$ | (Next Address)—Used to indicate that the external memory system is ready to accept a new bus cycle. |
| NMI | (Nonmaskable Interrupt)—Requests a nonmaskable interrupt as it did on the earlier versions of the microprocessor. |
| PCD | (Page Cache Disable Output)—Shows that the internal page caching is disabled by reflecting the state of the CR3 PCD bit. |
| $\overline{\text{PCHK}}$ | (Parity Check Output)—Signals a parity check error for data read from memory or I/O. |
| $\overline{\text{PEN}}$ | (Parity Enable Input)—Enables the machine check interrupt or exception. |
| PRDY | (Probe Ready Output)—Indicates that the probe mode has been entered for debugging. |
| PWT | (Page Write-Through Output)—Shows the state of the PWT bit in CR3. |
| $\overline{\text{R/S}}$ | (Request/Service)—This pin, provided for use with the Intel Debugging Port, causes an interrupt. |

| | |
|---|---|
| RESET | (Reset)—Initializes the Pentium, causing it to begin executing software at memory location FFFFFFF0H. The Pentium is reset to the real mode and the left-most 12 address connections remain logic 1 (FFFH) until a far jump or far call is executed. This allows compatibility with earlier microprocessors. Refer to Table 14–1 for the state of the Pentium after a hardware reset. |
| SCYC | (Split Cycle Output)—Signals a misaligned LOCKed bus cycle. |
| $\overline{\text{SMI}}$ | (System Management Interrupt Input)—Causes the Pentium to enter the system management mode of operation. |
| $\overline{\text{SMIACT}}$ | (System Management Interrupt Active Output)—Shows that the Pentium is operating in the system management mode. |
| TCK | (Testability Clock Input)—Selects the clocking function in accordance to the IEEE 1149.1 boundary scan interface. |
| TDI | (Test Data Input)—Input pin for test data clocked into the Pentium with the TCK signal. |
| TDO | (Test Data Output)—Data output pin used to gather test data and instructions shifted out of the Pentium with TCK. |
| TMS | (Test Mode Select Input)—Controls the operation of the Pentium in test mode. |
| TRST | (Test Reset Input)—Allows the test mode to be reset. |

**TABLE 14-1** State of the Pentium after a RESET

| *Register* | *Value (RESET)* | *VALUE (RESET+BIST)* |
|---|---|---|
| EAX | 0 | 0 (if test passes) |
| EDX | 0500XXXX | 0500XXXX |
| ECX,EBX,ESP, EBP,ESI,EDI | 0 | 0 |
| EFLAGS | 2 | 2 |
| EIP | 0000FFF0 | 0000FFF0 |
| CS | F000 | F000 |
| DS,ES,FS, GS,SS | 0 | 0 |
| DTR,TSS | 0 | 0 |
| CR0 | 60000010 | 60000010 |
| CR2,3,4 | 0 | 0 |
| DR0–3 | 0 | 0 |
| DR6 | FFFF0FF0 | FFFF0FF0 |
| DR7 | 00000400 | 00000400 |

*Note*: BIST = built-in self test. XXXX = Pentium version number.

| | |
|---|---|
| W/$\overline{R}$ | (Write/$\overline{Read}$)—Indicates that the current bus cycle is a write when a logic 1, or a read when a logic 0. |
| WB/$\overline{WT}$ | (Write-Back/Write-Through)—Selects the operation for the Pentium data cache. |

## The Memory System

The memory system for the Pentium microprocessor is 4G bytes in size, just like the 80386DX and 80486 microprocessors. The difference lies in the width of the memory data bus. The Pentium uses a 64-bit data bus to address memory, organized in 8 banks that each contain 512M bytes of data. Refer to Figure 14–2 for the organization of the Pentium physical memory system.

The memory system is divided into 8 banks that each store a byte of data with a parity bit. The Pentium, like the 80486, employs internal parity generation and checking logic for the memory system data bus information. The 64-bit-wide memory is important to double-precision floating-point data. Recall that a double-precision floating-point number is 64 bits in width. As with earlier Intel microprocessors, the memory system is numbered in bytes from byte 00000000H to byte FFFFFFFFH.

Memory selection is accomplished with the bank enable signals ($\overline{BE0}$–$\overline{BE7}$). These separate memory banks allow the Pentium to access any single byte, word, doubleword, or quadword with one memory transfer cycle. As with earlier memory selection logic, we often generate eight separate write strobes for writing to the memory system.

A new feature added to the Pentium is its ability to check and generate parity for the address bus (A31–A5) during certain operations. The AP pin provides the system with parity information, and the $\overline{APCHK}$ indicates a bad parity check for the address bus. The Pentium takes no action when an address parity error is detected. The error must be assessed by the system and the system must take appropriate action (an interrupt), if necessary.

How is a 32-bit memory system connected to the Pentium? The Pentium can function with a 32-bit-wide memory system by using a multiplexer to convert the 64-bit data bus to a 32-bit data bus. Figure 14–3 shows a set of bidirectional multiplexers (bidirectional buffers used as multiplexers) used to convert the Pentium's 64-bit data bus into a 32-bit data bus. Care must be taken when using this arrangement, because software could access a doubleword that crosses the boundary between the lower and upper halves of the data bus. All doublewords must be stored at doubleword boundaries. Note that a doubleword boundary is an address that is divisible by 4.

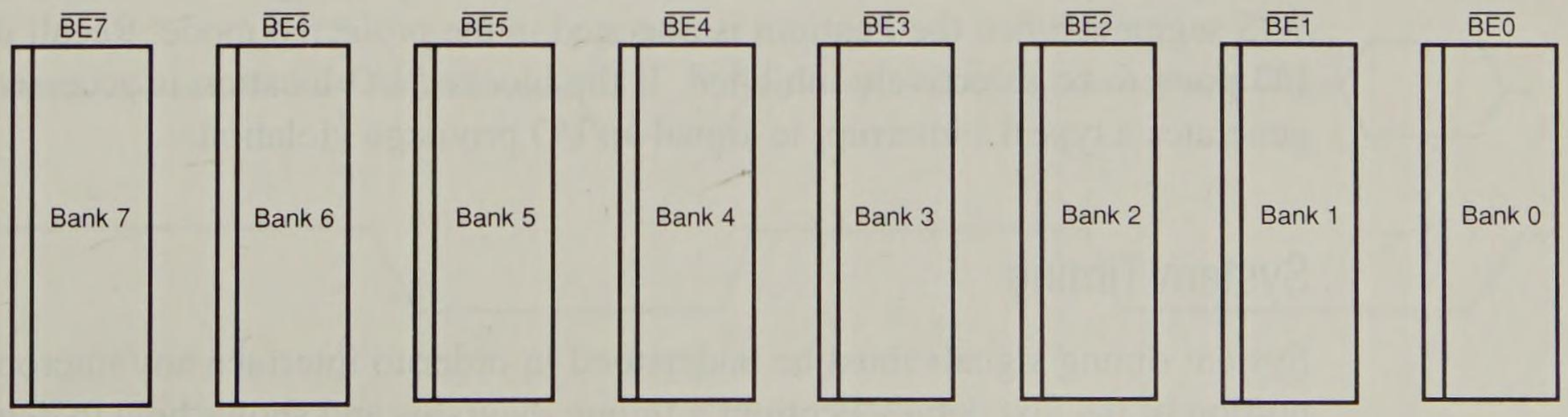



**FIGURE 14–2** The 8-byte-wide memory banks of the Pentium microprocessor.

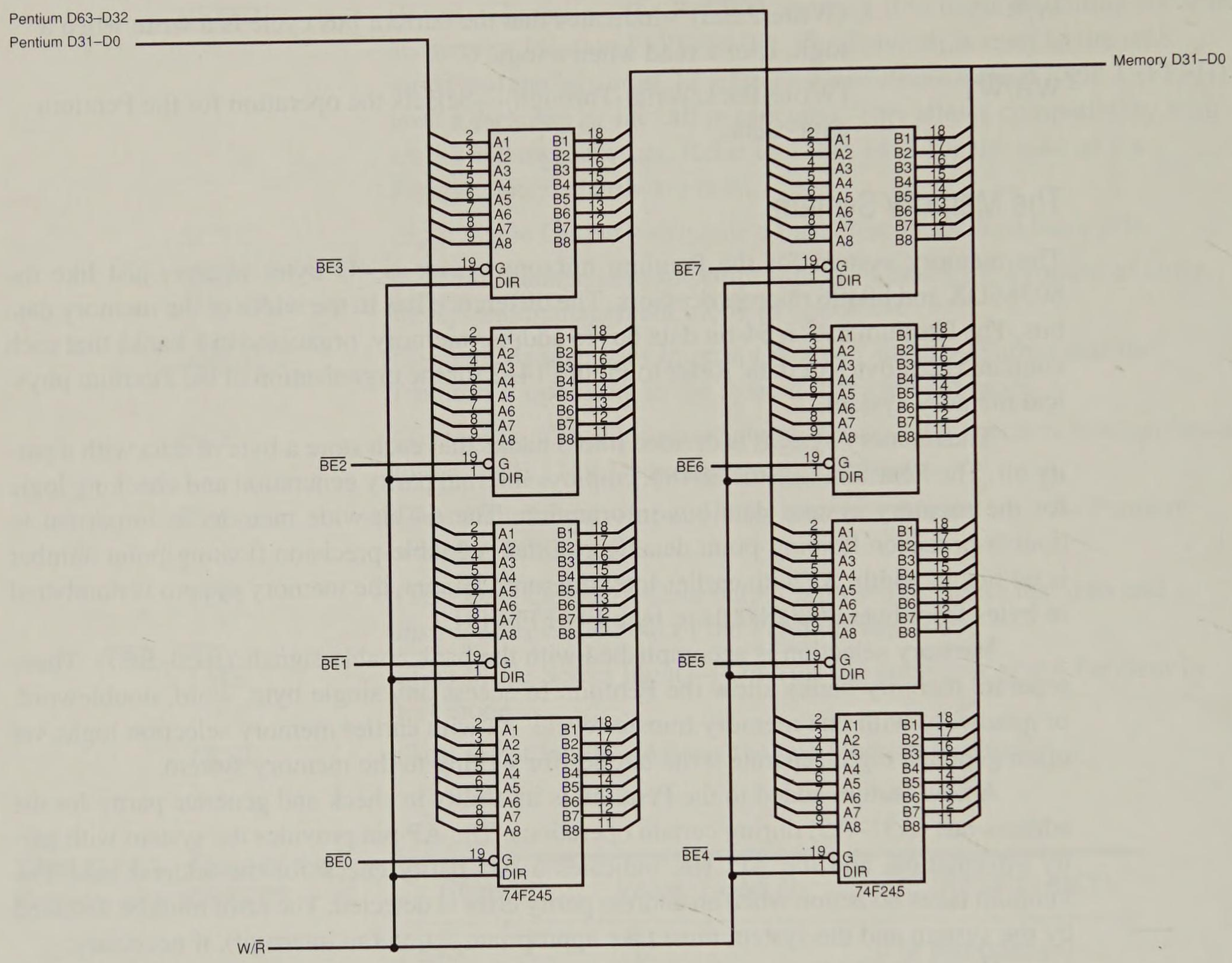



**FIGURE 14–3** A circuit that generates a 32-bit memory data bus from the 64-bit Pentium data bus.

## Input/Output System

The input/output system of the Pentium is completely compatible with earlier Intel microprocessors. The I/O port number appears on address lines A3–A15 with the bank enable signals used to select the actual memory banks used for the I/O transfer.

Beginning with the 80386 microprocessor, I/O privilege information is added to the TSS segment when the Pentium is operated in the protected mode. Recall that this allows I/O ports to be selectively inhibited. If the blocked I/O location is accessed, the Pentium generates a type 13 interrupt to signal an I/O privilege violation.

## System Timing

System timing signals must be understood in order to interface any microprocessor. This portion of the text details Pentium's timing diagrams and shows how to determine memory access times.

The basic nonpipelined Pentium memory cycle consists of two clocking periods: T1 and T2. Refer to Figure 14–4 for the basic nonpipelined read cycle. Notice from the timing diagram that the 66MHz Pentium is capable of 33 million memory transfers per second. This assumes that the memory can operate at that speed.

Also notice from the timing diagram that the W/$\overline{R}$ signal becomes valid if $\overline{ADS}$ is a logic zero at the positive edge of the clock (end of T1). This clock must be used to qualify the cycle as a read or a write.

During T1, the microprocessor issues the $\overline{ADS}$, W/$\overline{R}$, address, and M/$\overline{IO}$ signals. In order to qualify the W/$\overline{R}$ signal and generate appropriate $\overline{MRDC}$ and $\overline{MWTC}$ signals, we use a flip-flop to generate the W/$\overline{R}$ signal. Then we use a 2-line-to-1-line multiplexer to generate the memory and I/O control signals. Refer to Figure 14–5 for a circuit that generates the memory and I/O control signals for the Pentium microprocessor.

The data bus is sampled in synchronization with the positive transition of the clock pulse at the end of T2. The setup time before the clock is given as 3.8 ns, and the hold time after the clock is given as 2.0 ns. This means that the data window around the clock is 5.8 ns. The address appears on the 8.0 ns maximum after the start of T1. This means that the Pentium microprocessor operating at 66 MHz allows 30.3 ns (two clocking periods) minus the address delay time of 8.0 ns, minus the data setup time of 3.8 ns. Memory access time without any wait states is 30.3 – 8.0 – 3.8, or 18.5 ns. This is enough time to allow access to a SRAM but not to any DRAM, without inserting wait states into the timing.

Wait states are inserted into the timing by controlling the $\overline{BRDY}$ input to the Pentium. The $\overline{BRDY}$ signal must become a logic 0 by the end of T2, or additional T2 states are inserted into the timing. Refer to Figure 14–6 for a read cycle timing diagram that con-

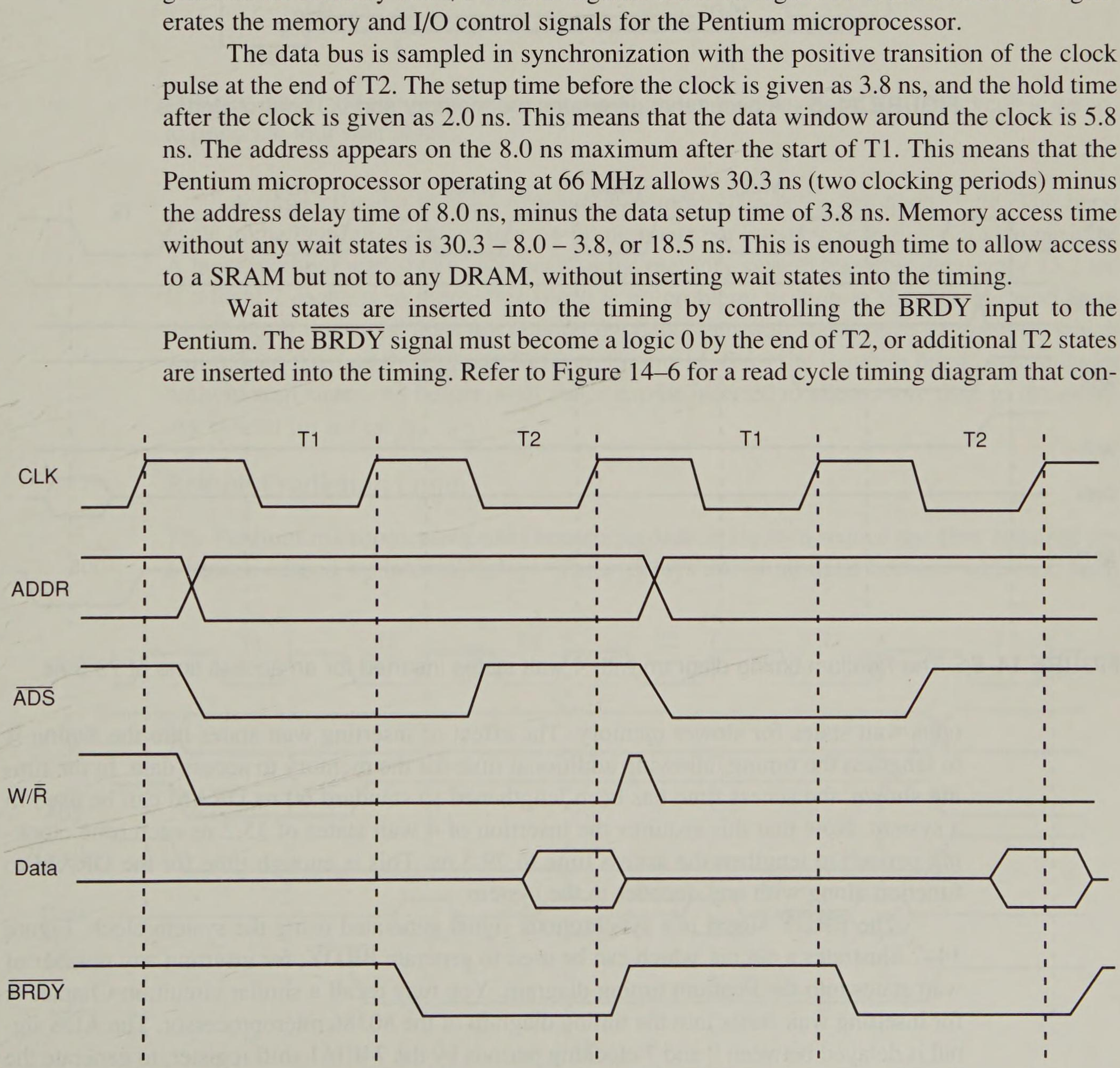



**FIGURE 14–4** The nonpipelined read cycle for the Pentium microprocessor.

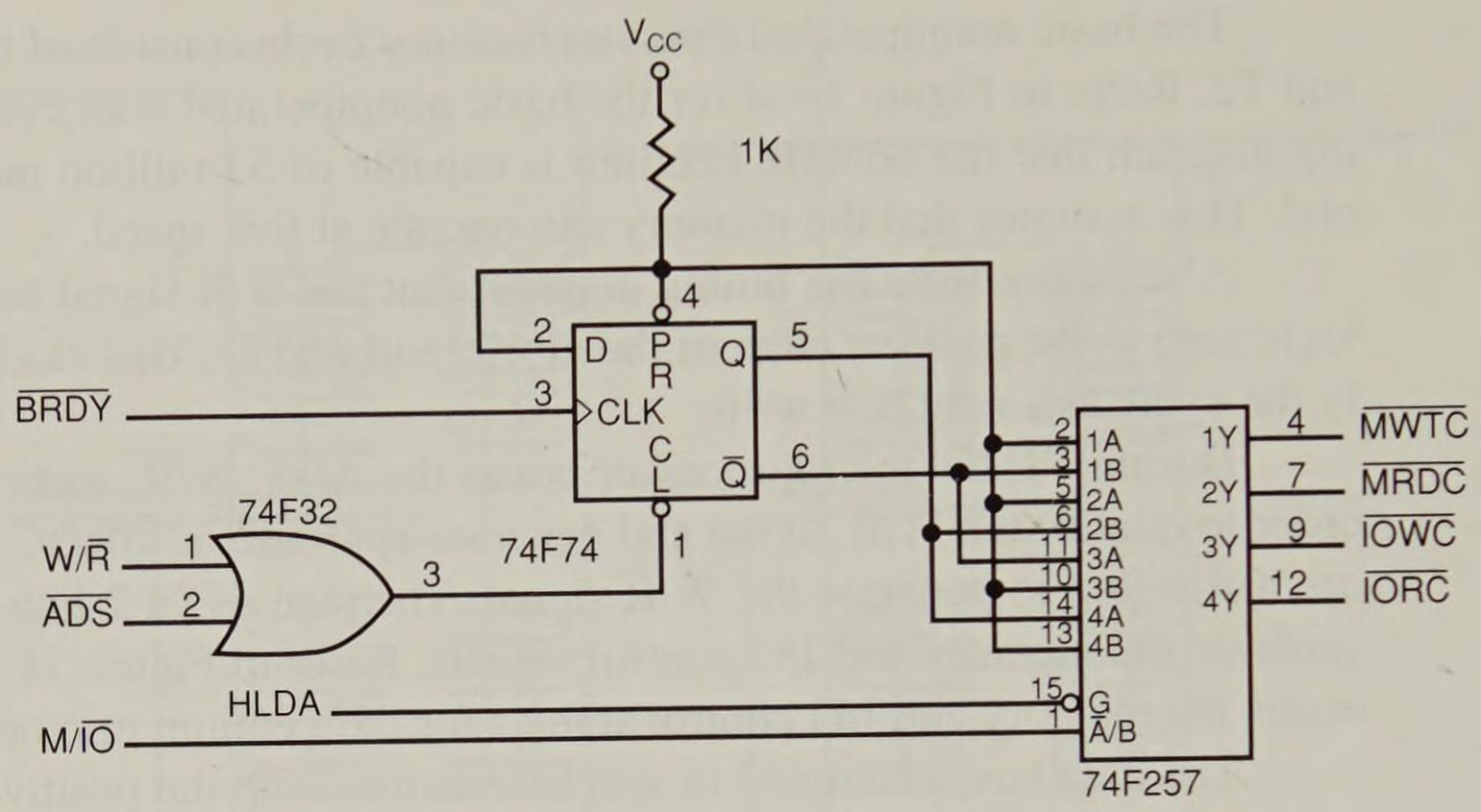



**FIGURE 14–5** A circuit that generates the memory and I/O control signals.

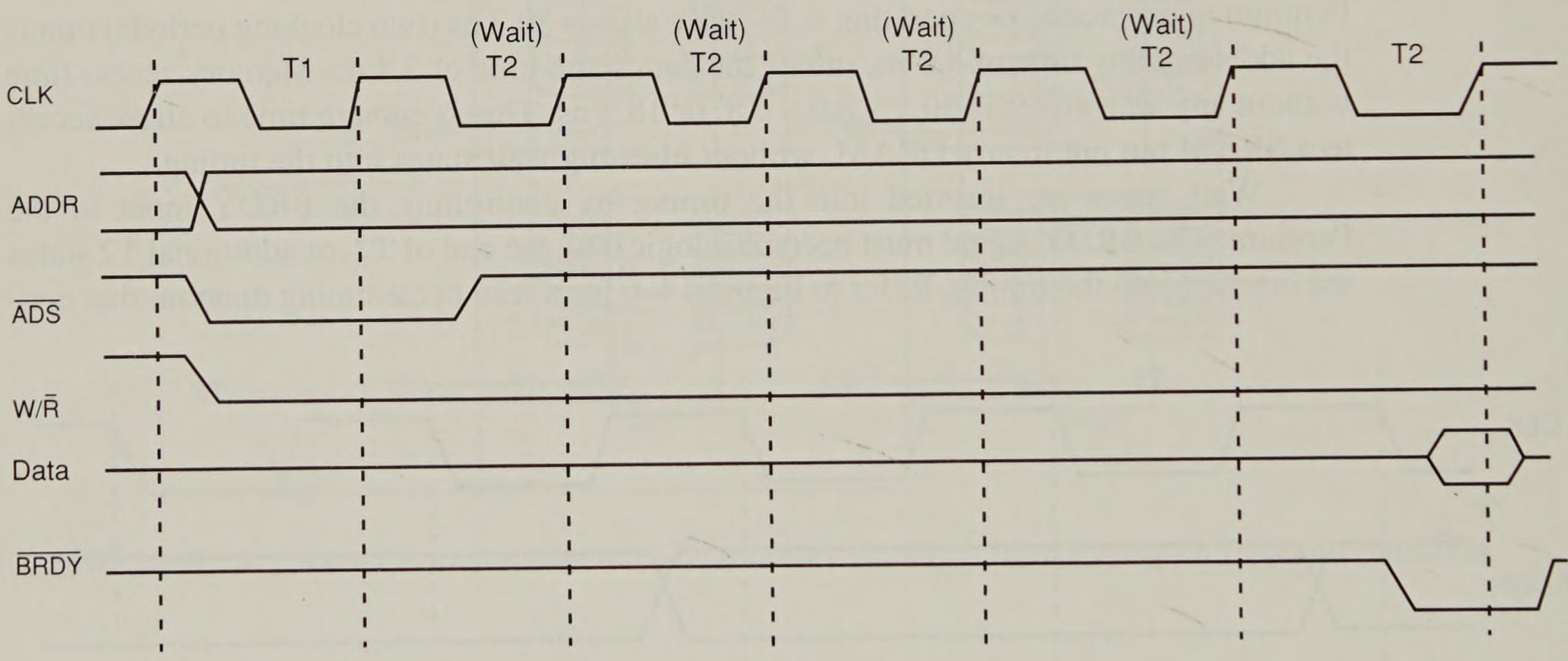



**FIGURE 14–6** The Pentium timing diagram with 4 wait states inserted for an access time of 79.5 ns.

tains wait states for slower memory. The effect of inserting wait states into the timing is to lengthen the timing, allowing additional time for the memory to access data. In the timing shown, the access time has been lengthened so standard 60 ns DRAM can be used in a system. Note that this requires the insertion of 4 wait states of 15.2 ns each (one clocking period) to lengthen the access time to 79.5 ns. This is enough time for the DRAM to function along with any decoder in the system.

The $\overline{BRDY}$ signal is a synchronous signal generated using the system clock. Figure 14–7 illustrates a circuit, which can be used to generate $\overline{BRDY}$, for inserting any number of wait states into the Pentium timing diagram. You may recall a similar circuit, in Chapter 9, for inserting wait states into the timing diagram of the 80386 microprocessor. The $\overline{ADS}$ signal is delayed between 0 and 7 clocking periods by the 74F161 shift register, to generate the $\overline{BRDY}$ signal. The exact number of wait states is selected by the 74F151 8-line-to-1-line multiplexer. In this example, the multiplexer selects the 4-wait output from the shift register.

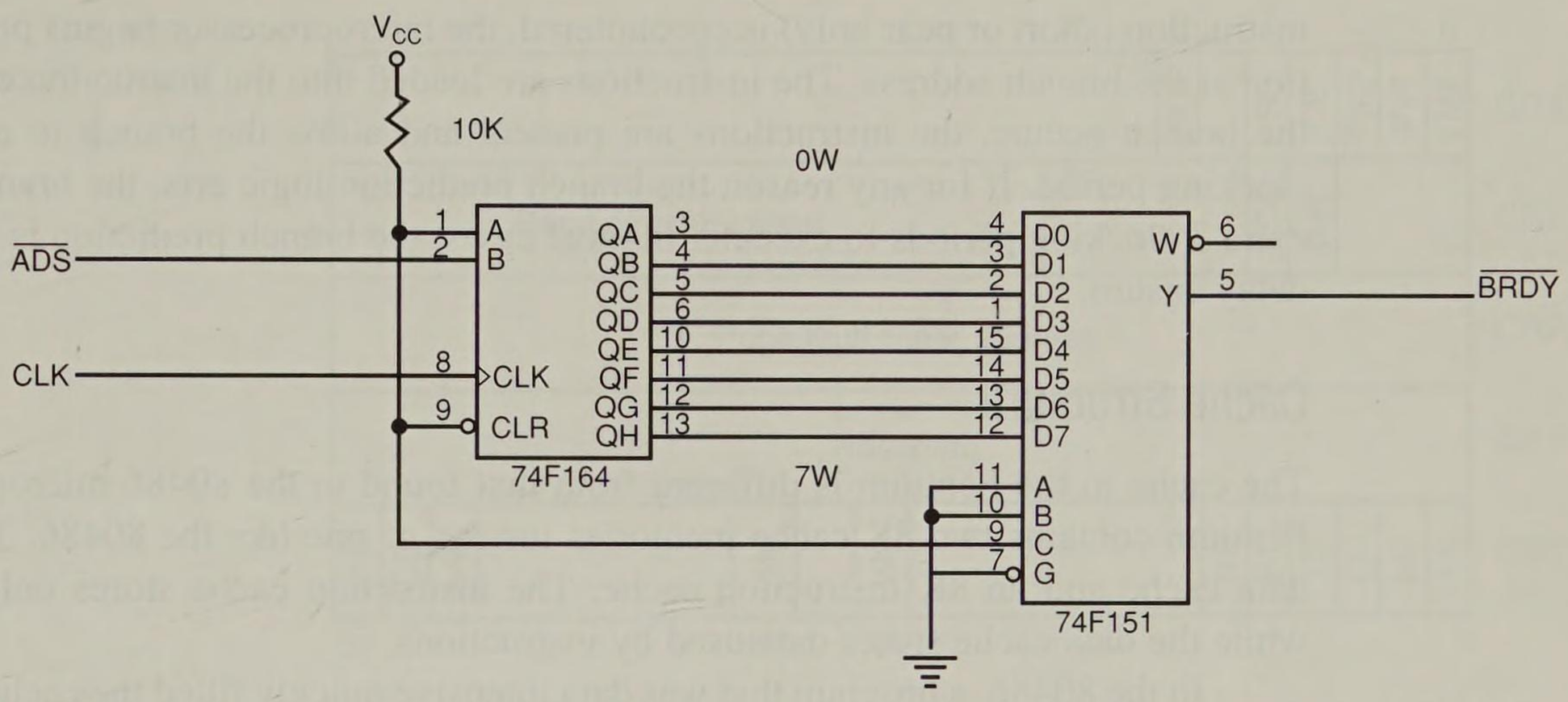



**FIGURE 14–7** A circuit that generates wait states by delaying $\overline{ADS}$. This circuit is wired to generate four wait states.

A more efficient method of reading memory data is via the burst cycle. The burst cycle in the Pentium transfers four 64-bit numbers per burst cycle in five clocking periods. A burst, without wait states, requires that the memory system transfers data every 15.2 ns. If a level 2 cache is in place, this speed is no problem, as long as the data are read from the cache. If the cache does not contain the data, then wait states must be inserted, which will reduce the data throughput. Refer to Figure 14–8 for the Pentium burst cycle transfer without wait states. As before, wait states can be inserted to allow more time to the memory system for accesses.

## Branch Prediction Logic

The Pentium microprocessor uses branch prediction logic to reduce the time required for a branch caused by internal delays. These delays are minimized because when a branch

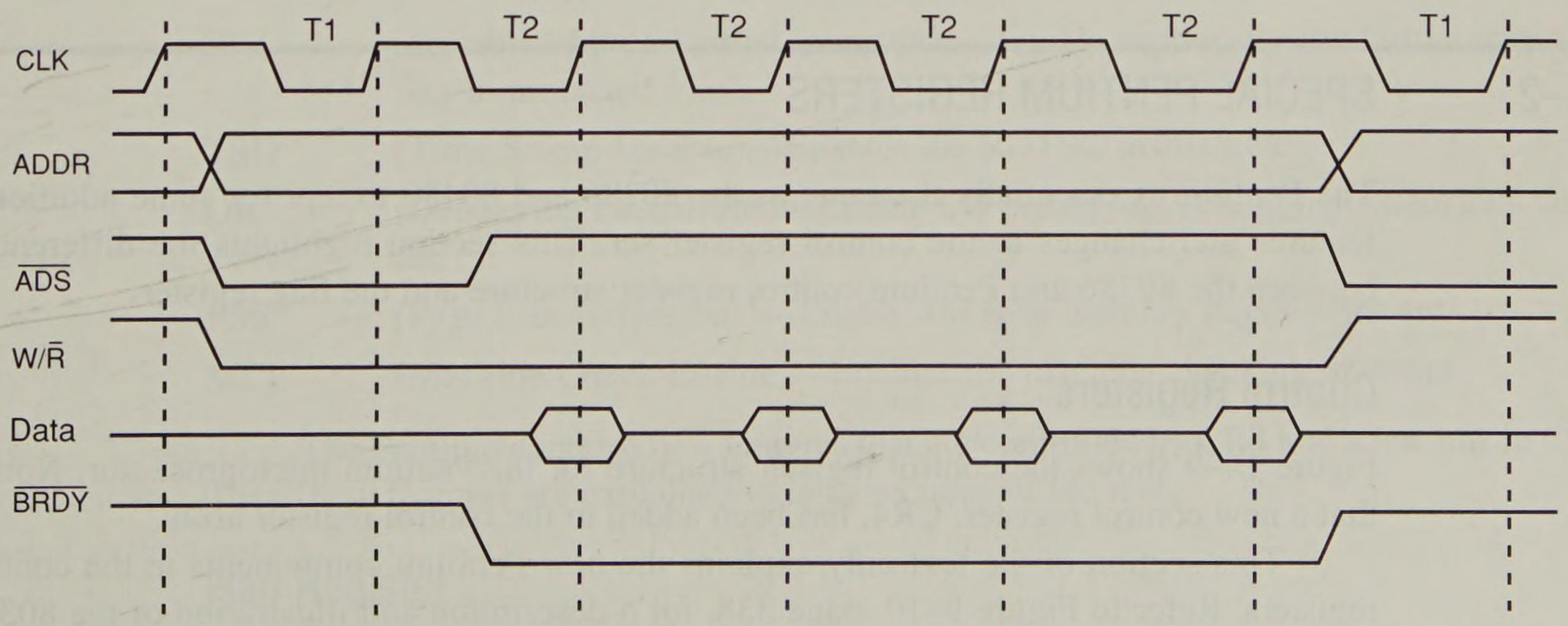



**FIGURE 14–8** The Pentium burst cycle operation that transfers four 64-bit data between the microprocessor and memory.

instruction (short or near only) is encountered, the microprocessor begins prefetch instruction at the branch address. The instructions are loaded into the instruction cache, so when the branch occurs, the instructions are present and allow the branch to execute in one clocking period. If for any reason the branch prediction logic errs, the branch requires an extra 3 clocking periods to execute. In most cases, the branch prediction is correct and no delay ensues.

### Cache Structure

The cache in the Pentium is different from that found in the 80486 microprocessor. The Pentium contains two 8K cache memories instead of one like the 80486. There is an 8K data cache and an 8K instruction cache. The instruction cache stores only instructions, while the data cache stores data used by instructions.

In the 80486, a program that was data intensive quickly filled the cache, allowing little room for instructions. This slowed the execution speed of the 80486 microprocessor. This cannot occur in the Pentium, because of the separate instruction cache.

### Superscaler Architecture

The Pentium microprocessor is organized with three execution units. One executes floating-point instructions, and the other two (U-pipe and V-pipe) execute integer instructions. This means that it is possible to execute three instructions simultaneously. For example, the FADD ST,ST(2) instruction, MOV EAX,10H instruction, and MOV EBX,12H instruction can all execute simultaneously, because none of these instructions depend on each other.

Software can be written to take advantage of this feature, by looking at the instructions in a program and modifying them when dependent instructions can be separated by nondependent instructions. These changes can result in a 40 percent improvement in execution speed in some software. Make sure that any new compiler or other application package takes advantage of this new superscaler feature of the Pentium.

## 14–2 SPECIAL PENTIUM REGISTERS

The Pentium is essentially the same as the 80386 and 80486 except for some additional features and changes to the control register set. This section highlights the differences between the 80386 and Pentium control register structure and the flag register.

### Control Registers

Figure 14–9 shows the control register structure for the Pentium microprocessor. Notice that a new control register, CR4, has been added to the control register array.

This section of the text only explains the new Pentium components in the control registers. Refer to Figure 9–10, page 338, for a description and illustration of the 80386 control registers. Following is a description of the new control bits and new control register, CR4:

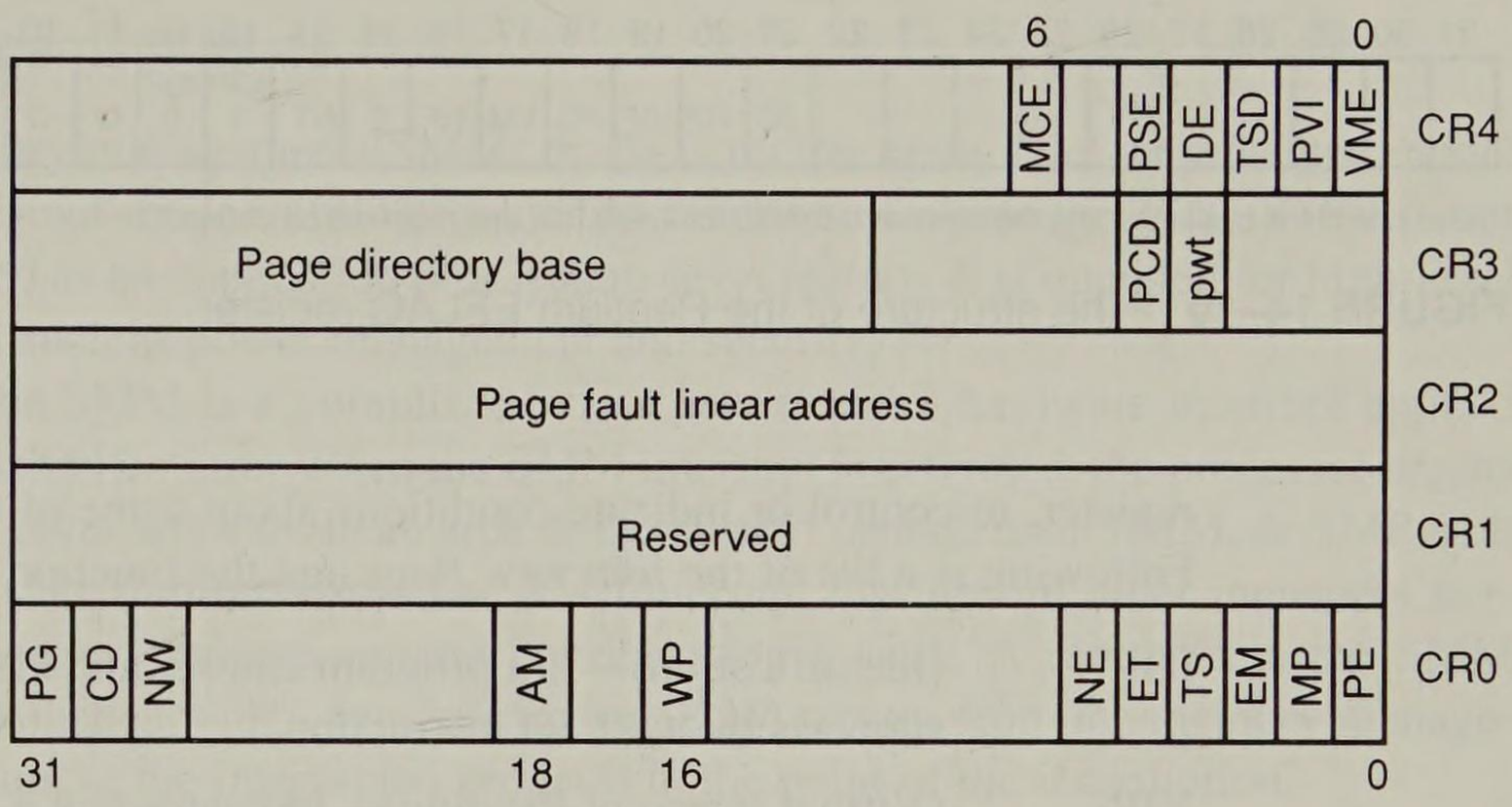



**FIGURE 14–9** The structure of the Pentium control registers.

CD (Cache Disable)—Enables the internal cache. If CD = 1, the cache will not fill with new data for cache misses, but it will continue to function for cache hits. If CD = 0, misses will cause the cache to fill with new data.

NW (Not Write-Through)—Selects the mode of operation for the data cache. If NW = 1, the data cache is inhibited from cache write-throughs.

AM (Alignment Mask)—Enables alignment checking when set. Note that alignment checking only occurs for protected mode operation when the user is at privilege level 3.

WP (Write Protect)—Protects user-level pages against supervisor-level write operations. When WP = 1, the supervisor can write to user-level segments.

NE (Numeric Error)—Enables standard numeric coprocessor error detection. If NE = 1, the $\overline{\text{FERR}}$ pin becomes active for a numeric coprocessor error. If NE = 0, any coprocessor error is ignored.

VME (Virtual Mode Extensions)—Enable support for the virtual interrupt flag in protected mode. If VME = 0, virtual interrupt support is disabled.

PVI (Protected Mode Virtual Interrupts)—Enable support for the virtual interrupt flag in protected mode.

TSD (Time Stamp Disable)—Enables the RDTSC instruction.

DE (Debugging Extensions)—Enable I/O breakpoint debugging extensions when set.

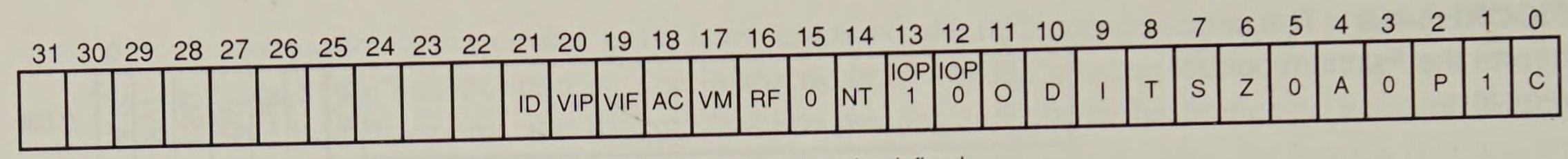
PSE (Page Size Extensions)—Enable 4M byte memory pages when set.

MCE (Machine Check Enable)—Enables the machine checking interrupt.

The Pentium contains new features that are controlled by CR4 and a few bits in CR0. These new features are explained in later sections of the text.

## Flag Register

The flag register has been changed in the Pentium microprocessor. Figure 14–10 pictures the contents of the EFLAG register. Notice that four new flag bits have been added to this




Note: The blank bits in the flag register are reserved for future use and must not be defined.

**FIGURE 14–10** The structure of the Pentium EFLAG register.

register, to control or indicate conditions about some of the new features in the Pentium. Following is a list of the four new flags and the function of each:

ID (Identification)—If a program can set and clear the ID flag, the processor supports the CPUID instruction.

VIP (Virtual Interrupt Pending)—Indicates that a virtual interrupt is pending.

VIF (Virtual Interrupt)—Is the image of the virtual interrupt flag IF, used with VIP.

AC (Alignment Check)—Indicates the state of the AM bit in control register 0.

### Built-In Self Test (BIST)

The built-in self test (BIST) is accessed on power-up by placing a logic 1 on INIT while the RESET pin changes from 1 to 0. The BIST tests 70 percent of the internal structure of the Pentium in approximately 150 μs. Upon completion of the BIST, the Pentium reports the outcome in register EAX. If EAX = 0, the BIST passed and the Pentium is ready for operation. If EAX contains any other value, the Pentium has malfunctioned and is faulty.

## 14–3 PENTIUM MEMORY MANAGEMENT

The memory management unit within the Pentium is compatible with the 80386 and 80486 microprocessors. Many of the features of these earlier microprocessors are basically unchanged in the Pentium. The main change is in the paging unit and a new system memory management mode.

### Paging Unit

The paging mechanism functions with 4K byte memory pages, or with a new extension available to the Pentium with 4M byte memory pages. As detailed in Chapter 9, the paging table structure can become large in a system that contains a large memory. Recall that to fully repage 4G bytes of memory, the microprocessor required slightly over 4M bytes of memory for just the page tables. In the Pentium, with the new 4M byte paging feature, this is dramatically reduced to just a single page table. The new 4M byte page sizes are selected by the PSE bit in control register 0.

Since the root page directory contains 1,024 entries, selecting 4M byte pages allows a single root directory, 4K bytes in length, to repage the entire memory system. Additional detail on 4M byte paging is not available from Intel at this time.

## Memory Management Mode

The system memory management (SMM) mode is on the same level as protected mode, real mode, and virtual mode, but is provided to function as a manager. The SMM is not intended to be used as an application or a system-level feature. It is intended for high-level system functions, such as power management and security.

Access to the SMM is accomplished via a new external hardware interrupt applied to the $\overline{\text{SMI}}$ pin on the Pentium. When the SMM interrupt is activated, the processor begins executing system-level software in an area of the system management RAM, or SMRAM, called the SMM state dump record. The SMI interrupt disables all other interrupts normally handled by user applications and the operating system. A return from the SMM interrupt is accomplished with a new instruction. RSM returns from the memory management mode interrupt to the interrupted program at the point of the interruption.

The SMM interrupt causes software initially stored at memory location 38000H to execute, because CS = 3000H and EIP = 8000H. This initial state can be changed using a jump to any location within the first 1M byte of memory. An environment similar to real-mode memory addressing is entered by the management mode interrupt. But instead of being able to address the first 1M of memory, as in real-mode addressing, SMM mode allows the Pentium to treat the memory system as a flat, 4G byte system.

In addition to executing software that begins at location 38000H, the SMM interrupt stores the state of the Pentium in what is called a dump record. The dump record is stored at memory locations 3FFA8H through 3FFFFH, with locations 3FE00H through 3FEF7H reserved by Intel. Table 14–2 lists the contents of the dump record.

The halt auto and I/O trap restarts are used when the SMM mode is exited by the RSM instruction. These data allow the RSM instruction to return to the halt state or the interrupt I/O instruction. If neither a halt nor an I/O operation were in effect upon entering the SMM mode, the RSM instruction reloads the state of the machine from the state dump and returns to the point of interruption.

The SMM mode can be used by the system before the normal operating system is placed in the memory and executed. It can also periodically be used to manage the system, provided normal software doesn't exist at locations 38000H–3FFFFH. If the system relocates the SMRAM before booting the normal operating system, it becomes available for use in addition to the normal system.

IN SMM mode the base address of the SMRAM is changed by modifying the value in the state dump base address register (locations 3FEF8H through 3F3FBH) after the first memory management mode interrupt. When the first RSM instruction is executed, returning control to the interrupted system, the new value from these locations changes the base address of the SMM interrupt for all future uses. For example, if the state dump base address is changed to 000E8000H, all subsequent SMM interrupts use locations E8000H–EFFFFH for the Pentium state dump. These locations are compatible with DOS and Windows.

# 14–4 NEW PENTIUM INSTRUCTIONS

The Pentium only contains one new instruction that functions with normal system software. The remainder of the new instructions are added to control the memory management

**TABLE 14–2** Pentium SMM state dump record

| *Location* | *Register* |
|---|---|
| 3FFFCH | CR0 |
| 3FFF8H | CR3 |
| 3FFF4H | EFLAGS |
| 3FFF0H | EIP |
| 3FFECH | EDI |
| 3FFE8H | ESI |
| 3FFE4H | EBP |
| 3FFE0H | ESP |
| 3FFDCH | EBX |
| 3FFD8H | EDX |
| 3FFD4H | ECX |
| 3FFD0H | EAX |
| 3FFCCH | DR6 |
| 3FFC8H | DR7 |
| 3FFC4H | TR |
| 3FFC0H | LDTR |
| 3FFBCH | GS |
| 3FFB8H | FS |
| 3FFB4H | DS |
| 3FFB0H | SS |
| 3FFACH | CS |
| 3FFA8H | ES |
| 3FF04H–3FFA7H | Reserved |
| 3FF02H | Halt auto restart |
| 3FF00H | I/O trap restart |
| 3FEFCH | SMM revision ID |
| 3FEF8H | State dump base |

mode feature and serializing instructions. Table 14–3 lists the new instructions added to the Pentium instruction set.

The CMPXCHG8B instruction is an extension of the CMPXCHG instruction added to the 80486 instruction set. The CMPXCHG8B instruction compares the 64-bit number stored in EDX and EAX with the contents of a 64-bit memory location or register pair. For

**TABLE 14–3** New Pentium instructions

| *Instruction* | *Function* |
|---|---|
| CMPXCHG8B | Compare and exchange eight bytes |
| CPUID | Returns the CPU identification code |
| RDTSC* | No information available for this instruction |
| RDMSR | Read model-specific register |
| WRMSR | Write model-specific register |
| RSM | Return from system interrupt |

**Note*: Intel reserves this instruction as proprietary.

**TABLE 14–4** CPUID instruction execution

| *Input Value (EAX)* | *Result* |
|---|---|
| 0 | EAX = 1 for the Pentium<br>EBX = 756E6547H<br>EDX = 49656E69H<br>EBX = 6C65746EH |
| 1 | EAX (bits 3–0) = Stepping ID<br>EAX (bits 7–4) = Model<br>EAX (bits 11–8) = Family<br>EAX (bits 31–12) = Reserved |

example, the CMPXCHG8B DATA1 instruction compares the eight bytes stored in memory location DATA1 with the 64-bit number in EDX and EAX. If DATA1 equals EDX:EAX, the 64-bit number stored in ECX:EBX is stored in memory location DATA1. If they are not equal, the contents of DATA1 are stored into EDX:EAX. Note that the zero flag bit indicates that the contents of EDX:EAX are equal or not equal to DATA1.

The CPUID instruction reads the CPU identification code and other information from the Pentium. Table 14–4 shows different information returned from the CPUID instruction for various input values of EAX. To use the CPUID instruction, first load EAX with the input value, and then execute CPUID. The information is returned in the registers indicated in the table.

The RDMSR and WRMSR instructions allow the model-specific registers to be read or written. The model-specific registers are unique to the Pentium and are used to trace, check performance, test, and check for machine errors.

The value in ECX accesses certain model-specific registers within the Pentium. Never use an undefined value in ECX before using the RDMSR or WRMSR instructions. If ECX = 0 before the read or write machine-specific register instruction, the value returned in EDX:EAX is the machine check address. If ECX = 1, the value is the machine check type, and if ECX = E, test register 12 (TR12) is accessed. Note that these are internal registers designed for in-house testing. The contents of these registers are proprietary to Intel.

The RMS instruction returns from a memory management mode interrupt. The memory management mode interrupt is explained in Section 14–3.

## 14–5 SUMMARY

1. The Pentium microprocessor is almost identical to the earlier 80386 and 80486 microprocessors. The main difference is that the Pentium has been internally modified so it contains a dual cache (instruction and data) and a dual integer unit. The Pentium also operates at a higher clock speed of 66 MHz.
2. The 66 MHz Pentium requires 3.3 A current, and the 60 MHz version requires 2.91 A current. The power supply must be a +5.0 V supply, with a regulation of ±5 percent.

3. The data bus on the Pentium is 64 bits in width and contains 8-byte-wide memory banks selected with bank enable signals $\overline{BE0}$–$\overline{BE7}$.
4. Memory access time, without wait states, is only about 15 μs in the 66 MHz Pentium. In many cases, this short access time requires wait states introduced by controlling the $\overline{BRDY}$ input to the Pentium.
5. The superscaler structure of the Pentium contains three independent processing units: a floating-point processor and two integer processing units, labeled U and V by Intel.
6. The cache structure of the Pentium is modified to include two caches. One 8K × 8 cache is designed as an instruction cache, and the other 8K × 8 cache is a data cache. The data cache can be operated as either a write-through or a write-back cache.
7. A new mode of operation called the system memory management (SMM) mode has been added to the Pentium. The SMM mode is accessed via the system memory management interrupt applied to the $\overline{SMI}$ input pin. In response to $\overline{SMI}$, the Pentium begins executing software at memory location 38000H.
8. New instructions include the CMPXCHG8B, RSM, RDMSR, WRMSR, and CPUID. The CMPXCHG8B instruction is similar to the 80486 CMPXCHG instruction. The RSM instruction returns from the system memory management interrupt. The RDMSR and WRMSR instructions read or write to the machine-specific registers. The CPUID instruction reads the CPU identification code from the Pentium.
9. The built-in self test (BIST) allows the Pentium to be tested when power is first applied to the system. A normal power-up reset activates the RESET input to the Pentium. A BIST power-up reset activates INIT and then deactivates the RESET pin. EAX is equal to a 00000000H if the BIST passes.
10. A new, proprietary Intel modification to the paging unit allows 4M byte memory pages instead of 4K byte pages.

## 14–6 QUESTIONS AND PROBLEMS

1. How much memory is accessible to the Pentium microprocessor?
2. The memory data bus width is _____ in the Pentium.
3. What is the purpose of the DP0–DP7 pins on the Pentium?
4. If the Pentium operates at 66 MHz, what frequency clock signal is applied to the CLK pin?
5. What is the purpose of the $\overline{BRDY}$ pin on the Pentium?
6. What is the purpose of the AP pin on the Pentium?
7. How much memory access time is allowed by the Pentium, without wait states, when it is operated at 66 MHz?
8. What Pentium pin is used to insert wait states into the timing?
9. A wait state is an extra T_____ clocking period.
10. Explain how two integer units allow the Pentium to execute two nondependent instructions simultaneously.
11. How many caches are found in the Pentium, and what are their sizes?
12. How wide is the Pentium memory data sample window for a memory read operation?

13. Can the Pentium execute three instructions simultaneously?
14. What is the purpose of the $\overline{\text{SMI}}$ pin?
15. What is the system memory management mode of operation for the Pentium?
16. How is the system memory management mode exited?
17. Where does the Pentium begin to execute software for an $\overline{\text{SMI}}$ interrupt input?
18. How can the system memory management unit dump address be modified?
19. Explain the operation of the CMPXCHG8B instruction.
20. What information is returned in register EAX after the CPUID instruction executes with an initial value of 0 in EAX?
21. What new flag bits are added to the Pentium microprocessor?
22. What new control register is added to the Pentium microprocessor?

# APPENDIX A

# The Assembler and Disk Operating System

This appendix is provided so the use of the assembler can be understood, and to show the DOS (disk operating system) and BIOS (basic I/O system) function calls that are used by assembly language to control the IBM-PC or its clone. The function calls control everything, from reading and writing disk data to managing the keyboard and displays. The assembler represented in this text is the Microsoft ML (version 6.X) and MASM (version 5.10) macro assembler programs.

## ASSEMBLER USAGE

The assembler program requires, first, that a symbolic program be written, using a word processor, text editor, or the workbench program provided with the assembler package. The editor provided with version 5.10 is M.EXE, a strictly full-screen editor. The editor provided with version 6.X is PWB.EXE, a fully integrated development system that contains extensive help. Refer to the documentation that accompanies your assembler package for details on the operation of the editor program. If at all possible, use version 6.X, because it contains a detailed help file that guides the user through assembly language statements, directives, and even the DOS and BIOS interrupt function calls.

If you are using a word processor to develop your software, make sure it is initialized to generate a pure ASCII file. The source file that you generate must use the extension .ASM, required for the assembler to properly identify your source program.

Once your source file is prepared, it must be assembled. If you are using the workbench provided with version 6.X, this is accomplished by selecting the compile feature with your mouse. If you are using a word processor and DOS command lines with version 5.10, see Example A–1 for the dialog for version 5.10 to assemble a file called FROG.ASM. This example shows the portions typed by the user in italics.

**EXAMPLE A–1**

```
A>MASM

Microsoft (R) Macro Assembler Version 5.10
```

```
Copyright (C) Microsoft Corp 1981, 1989.  All rights reserved.

Source filename [.ASM]:FROG
Object filename [FROG.OBJ]:FROG
List filename [NUL.LST]:FROG
Cross reference [NUL.CRF]:FROG
```

Once a program is assembled, it must be linked before it can be executed. The linker converts the object file into an executable file (.EXE). Example A–2 shows the dialog required for the linker using a MASM version 5.10 object file. If the ML version 6.X assembler is used, it automatically assembles and links a program using the COMPILE or BUILD command from workbench. After compiling with ML, workbench allows the program to be debugged with a debugging tool called code view. Code view is also available with MASM, but CV must be typed at the DOS command line to access it.

**EXAMPLE A–2**

```
A>LINK

Microsoft (R) Overlay Linker Version 3.64
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Object modules [.OBJ]:FROG
Run file [FROG.EXE]:FROG
List file [NUL.MAP]:FROG
Libraries [.LIB]:SUBR
```

Version 6.X of the Microsoft MASM program contains the programmer's workbench program. This program allows an assembly language program to be developed with its full-screen editor and tool bar. Figure A–1 illustrates the display found with programmer's workbench. To access this program, type PWB at the DOS prompt. The make option allows a program to be automatically assembled and linked, making these tasks simple in comparison to version 5.10 of the assembler.

## ASSEMBLER MEMORY MODELS

Memory models and the .MODEL statement are introduced in Chapter 6. Here we completely define the memory models available for software development. Each model defines the way a program is stored in the memory system. Table A–1 describes the different models available with MASM and also ML.

Note that the tiny model is used to create a .COM file instead of an execute file. The .COM file is different because all data and codes fit into one code segment. A .COM file must have the program originated to start at offset address 0100H. A .COM file loads from the disk and executes faster than the normal execute file (.EXE). For most applications, use the execute file (.EXE) and the small memory model.

When models are used to create a program, certain defaults apply, as illustrated in Table A–2. The directive in this table is used to start a particular type of segment for the models listed in the table. If the .CODE directive is placed in a program, it indicates the beginning of the code segment. Likewise, .DATA indicates the start of a data segment.

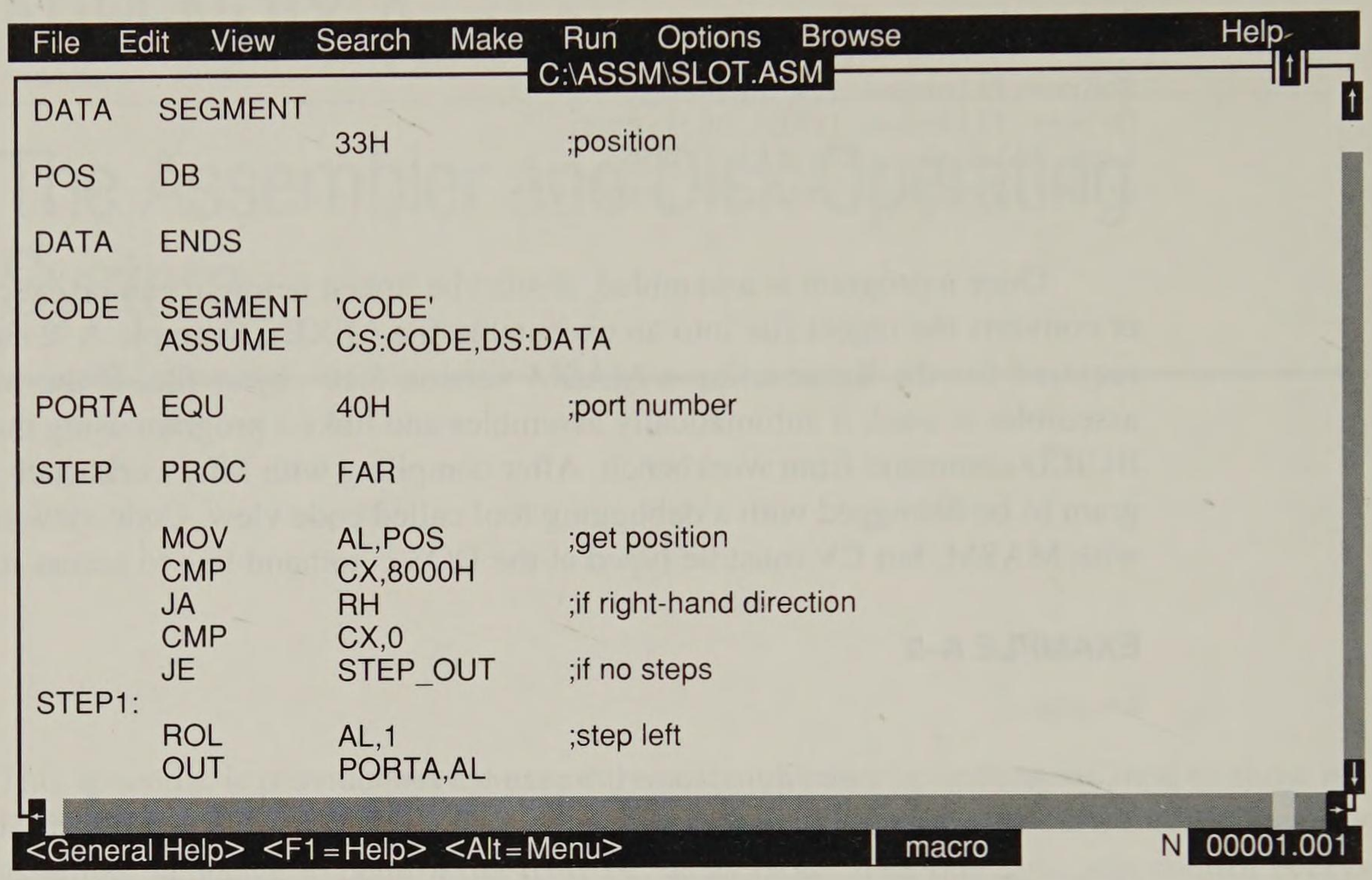



**FIGURE A–1** The edit screen from programmer's workbench used to develop assembly language programs.

**TABLE A–1** Memory models for the assembler

| *Type* | *Description* |
|---|---|
| Tiny | All data and code fit into one segment, the code segment. Tiny model programs are written in the .COM file format, which means that the program must be originated at memory location 0100H. This model is most often used with small programs. |
| Small | All data fit into a single 64K-byte data segment and all codes fit into another single 64K-byte code segment. This allows all codes to be accessed with near jumps and calls. |
| Medium | All data fit into a single 64K-byte data segment, and codes fit into more than one code segment. This allows codes to exist in multiple segments. |
| Compact | All codes fit into a single 64K-byte code segment, and data fit into more than one data segment. |
| Large | Both code and data fit into multiple code and data segments. |
| Huge | Same as large, but allows data segments that are larger than 64K bytes. |
| Flat | Not available in MASM version 5.10. The flat memory model uses one segment, with a maximum length of 512M bytes, to store data and codes. |

**TABLE A–2** Defaults for the .MODEL directive

| *Model* | *Directive* | *Name* | *Align* | *Combine* | *Class* | *Group* |
|---|---|---|---|---|---|---|
| Tiny | .CODE | _TEXT | Word | PUBLIC | 'CODE' | DGROUP |
| | .FARDATA | FAR_DATA | Para | Private | 'FAR_DATA' | |
| | .FARDATA? | FAR_BSS | Para | Private | 'FAR_BSS' | |
| | .DATA | _DATA | Word | PUBLIC | 'DATA' | DGROUP |
| | .CONST | CONST | Word | PUBLIC | 'CONST' | DGROUP |
| | .DATA? | _BSS | Word | PUBLIC | 'BSS' | DGROUP |
| Small | .CODE | _TEXT | Word | PUBLIC | 'CODE' | |
| | .FARDATA | FAR_DATA | Para | Private | 'FAR_DATA' | |
| | .FARDATA? | FAR_BSS | Para | Private | 'FAR_BSS' | |
| | .DATA | _DATA | Word | PUBLIC | 'DATA' | DGROUP |
| | .CONST | CONST | Word | PUBLIC | 'CONST' | DGROUP |
| | .DATA? | _BSS | Word | PUBLIC | 'BSS' | DGROUP |
| | .STACK | STACK | Para | STACK | 'STACK' | DGROUP |
| Medium | .CODE | name_TEXT | Word | PUBLIC | 'CODE' | |
| | .FARDATA | FAR_DATA | Para | Private | 'FAR_DATA' | |
| | .FARDATA? | FAR_BSS | Para | Private | 'FAR_BSS' | |
| | .DATA | _DATA | Word | PUBLIC | 'DATA' | DGROUP |
| | .CONST | CONST | Word | PUBLIC | 'CONST' | DGROUP |
| | .DATA? | _BSS | Word | PUBLIC | 'BSS' | DGROUP |
| | .STACK | STACK | Para | STACK | 'STACK' | DGROUP |
| Compact | .CODE | _TEXT | Word | PUBLIC | 'CODE' | |
| | .FARDATA | FAR_DATA | Para | Private | 'FAR_DATA' | |
| | .FARDATA? | FAR_BSS | Para | Private | 'FAR_BSS' | |
| | .DATA | _DATA | Word | PUBLIC | 'DATA' | DGROUP |
| | .CONST | CONST | Word | PUBLIC | 'CONST' | DGROUP |
| | .DATA? | _BSS | Word | PUBLIC | 'BSS' | DGROUP |
| | .STACK | STACK | Para | STACK | 'STACK' | DGROUP |
| Large or huge | .CODE | name_TEXT | Word | PUBLIC | 'CODE' | |
| | .FARDATA | FAR_DATA | Para | Private | 'FAR_DATA' | |
| | .FARDATA? | FAR_BSS | Para | Private | 'FAR_BSS' | |
| | .DATA | _DATA | Word | PUBLIC | 'DATA' | DGROUP |
| | .CONST | CONST | Word | PUBLIC | 'CONST' | DGROUP |
| | .DATA? | _BSS | Word | PUBLIC | 'BSS' | DGROUP |
| | .STACK | STACK | Para | STACK | 'STACK' | DGROUP |
| Flat | .CODE | _TEXT | Dword | PUBLIC | 'CODE' | |
| | .FARDATA | _DATA | Dword | PUBLIC | 'DATA' | |
| | .FARDATA? | _BSS | Dword | PUBLIC | 'BSS' | |
| | .DATA | _DATA | Dword | PUBLIC | 'DATA' | |
| | .CONST | CONST | Dword | PUBLIC | 'CONST' | |
| | .DATA? | _BSS | Dword | PUBLIC | 'BSS' | |
| | .STACK | STACK | Dword | PUBLIC | 'STACK' | |

The *name* column indicates the name of the segment. *Align* indicates whether the segment is aligned on a word, doubleword, or a 16-byte paragraph. *Combine* indicates the type of segment created. The *class* indicates the class of the segment, such as 'CODE' or 'DATA'. The *group* indicates the group type of the segment.

Example A–3 shows a program that uses the small model. The small model is used for programs that contain one DATA and one CODE segment. This applies to many programs. Notice that not just the program is listed, but so is all the information generated by the assembler. Here the .DATA directive and .CODE directive indicate the start of segments. Also notice how the DS register is loaded in this program.

**EXAMPLE A–3**

```
Microsoft (R) Macro Assembler Version 6.11


                        .MODEL SMALL
                        .STACK 100H
0000                    .DATA

0000 0A                 FROG    DB      10
0001 0064 [             DATA1   DB      100 DUP (2)
             02
                ]

0000                    .CODE

0000 B8 ---- R          BEGIN:  MOV     AX,DGROUP       ;set up DS
0003 8E D8                      MOV     DS,AX
                                .
                                .
                                .
                                END     BEGIN

Segments and Groups:

N a m e                 Size    Length  Align  Combine  Class

DGROUP . . . . . GROUP
_DATA  . . . . . 16 Bit  0065    Word   Public   'DATA'
STACK  . . . . . 16 Bit  0100    Para   Stack    'STACK'
_TEXT  . . . . . 16 Bit  0005    Word   Public   'CODE'

Symbols:

N a m e                 Type    Value   Attr

@CodeSize  . . . . . Number  0000h
@DataSize  . . . . . Number  0000h
@Interface . . . . . Number  0000h
@Model . . . . . Number  0002h
@code  . . . . . Text            _TEXT
@data  . . . . . Text            DGROUP
@fardata?  . . . . . Text        FAR_BSS
@fardata . . . . . Text          FAR_DATA
@stack . . . . . Text            DGROUP
BEGIN  . . . . . L Near  0000    _TEXT
DATA1  . . . . . Byte    0001    _DATA
```

```
FROG . . . . . . . . Byte    0000    _DATA

          0 Warnings
          0 Errors
```

Example A–4 lists a program that uses the large model. Notice how it differs from the small model program of Example A–3. Models can be very useful in developing software, but we often use full-segment descriptions, as depicted in Chapter 7.

**EXAMPLE A–4**

```
Microsoft (R) Macro Assembler Version 6.11

                        .MODEL LARGE
                        .STACK 1000H
0000                    .FARDATA?

0000 00                 FROG    DB      ?
0001 0064 [             DATA1   DW      100 DUP (?)
             0000
                  ]

0000                    .CONST

0000 54 68 69 73 20 69  MES1    DB      'This is a character string'
     73 20 61 20 63 68
     61 72 61 63 74 65
     72 20 73 74 72 69
     6E 67
001A 53 6F 20 69 73 20  MES2    DB      'So is this!'
     74 68 69 73 21

0000                    .DATA

0000 000C               DATA2   DW      12
0002 00C8 [             DATA3   DB      200 DUP (1)
             01
                ]

0000                    .CODE

0000                    FUNC    PROC    FAR
                                .
                                .
                                .
                                .
0000 CB                         RET

0001                    FUNC    ENDP

                                END     FUNC

Segments and Groups:

N a m e                 Size    Length Align  Combine  Class

DGROUP . . . . . . . . .GROUP
_DATA  . . . . . . . . .16 Bit  00CA   Word   Public   'DATA'
STACK  . . . . . . . . .16 Bit  1000   Para   Stack    'STACK'
```

```
CONST  . . . . . .16 Bit  0025   Word   Public   'CONST' ReadOnly
EXA_TEXT . . . . . .16 Bit  0001   Word   Public   'CODE'
FAR_BSS  . . . . . .16 Bit  00C9   Para   Private  'FAR_BSS'
_TEXT  . . . . . .16 Bit  0000   Word   Public   'CODE'

Procedures, parameters and locals:

N a m e                 Type    Value  Attr

FUNC . . . . . .P Far   0000   EXA_TEXT Length= 0001 Public

Symbols:

N a m e                 Type    Value  Attr

@CodeSize  . . . . . .Number  0001h
@DataSize  . . . . . .Number  0001h
@Interface . . . . . .Number  0000h
@Model . . . . . .Number  0005h
@code  . . . . . .Text          EXA_TEXT
@data  . . . . . .Text          DGROUP
@fardata?  . . . . . .Text          FAR_BSS
@fardata . . . . . .Text          FAR_DATA
@stack . . . . . .Text          DGROUP
DATA1  . . . . . .Word    0001   FAR_BSS
DATA2  . . . . . .Word    0000   _DATA
DATA3  . . . . . .Byte    0002   _DATA
FROG . . . . . .Byte    0000   FAR_BSS
MES1 . . . . . .Byte    0000   CONST
MES2 . . . . . .Byte    001A   CONST

          0 Warnings
          0 Errors
```

## DOS FUNCTION CALLS

**EXAMPLE A–5**

```
0000 B4 06              MOV     AH,6
0002 B2 41              MOV     DL,'A'
0004 CD 21              INT     21H
```

In order to use DOS function calls, place the function number into register AH and load all other pertinent information into registers as described in Table A–3. Once this is accomplished, follow with an INT 21H to execute the DOS function. Example A–5 shows how to display an ASCII A on the CRT screen at the current cursor position with a DOS function call. Table A–3 is a complete listing of the DOS function calls. Note that some function calls require a segment and offset address, indicated as DS:DI for example. This means the data segment is the segment address and DI is the offset address. All of the function calls use INT 21H, and AH contains the function call number. Note that functions marked with an @ should not be used unless DOS version 2.XX is in use. As a rule, DOS function calls save all registers not used as exit data, but in certain cases, some registers may change. In order to prevent problems, it is advisable to save registers where problems occur. (Note: Figures referenced in Table A–3 appear on pp. 670–672.)

**TABLE A–3** DOS function calls (pp. 643–670)

| **00H** | TERMINATE A PROGRAM |
|---|---|
| Entry | AH = 00H<br>CS = program segment prefix address |
| Exit | DOS is entered |
| **01H** | READ THE KEYBOARD |
| Entry | AH = 01H |
| Exit | AL = ASCII character |
| Notes | If AL = 00H, the function call must be invoked again to read an extended ASCII character. Refer to Table 6–1 for a listing of the extended ASCII keyboard codes. This function call automatically echoes whatever is typed to the video screen. |
| **02H** | WRITE TO STANDARD OUTPUT DEVICE |
| Entry | AH = 02H<br>AL = ASCII character to be displayed |
| Notes | This function call normally displays data on the video display. |
| **03H** | READ CHARACTER FROM COM1 |
| Entry | AH = 03H |
| Exit | AL = ASCII character read from the communications port |
| Notes | This function call reads data from the serial communications port. |

| **04H** | WRITE TO COM1 |
|---|---|
| Entry | AH = 04H<br>DL = character to be sent out of COM1 |
| Notes | This function transmits data through the serial communications port. |
| **05H** | WRITE TO LPT1 |
| Entry | AH = 05H<br>DL = ASCII character to be printed |
| Notes | Prints DL on the line printer attached to LPT1 |
| **06H** | DIRECT CONSOLE READ/WRITE |
| Entry | AH = 06H<br>DL = 0FFH or DL = ASCII character |
| Exit | AL = ASCII character |
| Notes | If DL = 0FFH on entry, then this function reads the console. If DL = ASCII character, then this function displays the ASCII character on the console video screen. If a character is read from the console keyboard, the zero flag (ZF) indicates whether a character was typed. A zero condition indicates no key is typed, and a not-zero condition indicates that AL contains the ASCII code of the key or a 00H. If AL = 00H, the function must again be invoked to read an extended ASCII character from the keyboard. Note that the key does not echo to the video screen. |
| **07H** | DIRECT CONSOLE INPUT WITHOUT ECHO |
| Entry | AH = 07H |
| Exit | AL = ASCII character |
| Notes | This functions exactly like function number 06H with DL = 0FFH, but it will not return from the function until the key is typed. |

| **08H** | READ STANDARD INPUT WITHOUT ECHO |
|---|---|
| Entry | AH = 08H |
| Exit | AL = ASCII character |
| Notes | Performs like function 07H, except it reads the standard input device. The standard input device can be assigned as either the keyboard or the COM port. This function also responds to a control-break, where function 06H and 07H do not. A control-break causes INT 23H to execute. |
| **09H** | DISPLAY A CHARACTER STRING |
| Entry | AH = 09H<br>DS:DX = address of the character string |
| Notes | The character string must end with an ASCII $ (24H). The character string can be of any length and may contain control characters, such as carriage return (0DH) and line feed (0AH). |
| **0AH** | BUFFERED KEYBOARD INPUT |
| Entry | AH = 0AH<br>DS:DX = address of keyboard input buffer |
| Notes | The first byte of the buffer contains the size of the buffer (up to 255). The second byte is filled with the number of characters typed upon return. The third byte, through the end of the buffer, contains the character string typed, followed by a carriage return (0DH). This function continues to read the keyboard (displaying data as typed) until either the specified number of characters are typed or a carriage return (enter) key is typed. |

| **0BH** | TEST STATUS OF THE STANDARD INPUT DEVICE |
|---|---|
| Entry | AH = 0BH |
| Exit | AL = status of the input device |
| Notes | This function tests the standard input device to determine if data are available. If AL = 00, no data are available. If AL = 0FFH, then data are available that must be input using function number 08H. |
| **0CH** | CLEAR KEYBOARD BUFFER AND INVOKE KEYBOARD FUNCTION |
| Entry | AH = 0CH<br>AL = 01H, 06H, 07H, or 0AH |
| Exit | See exit for functions 01H, 06H, 07H, or 0AH. |
| Notes | The keyboard buffer holds keystrokes while programs execute other tasks. This function empties or clears the buffer and then invokes the keyboard function located in register AL. |
| **0DH** | FLUSH DISK BUFFERS |
| Entry | AH = 0DH |
| Notes | Erases all file names stored in disk buffers. This function does not close the files specified by the disk buffers, so care must be exercised in its use. |
| **0EH** | SELECT DEFAULT DISK DRIVE |
| Entry | AH = 0EH<br>DL = desired default disk drive number |
| Exit | AL = the total number of drives present in the system |
| Notes | Drive A = 00H, drive B = 01H, drive C = 02H, and so forth. |

| **0FH** | @OPEN FILE WITH FCB |
|---|---|
| Entry | AH = 0FH<br>DS:DX = address of the unopened file control block (FCB) |
| Exit | AL = 00H if file found<br>AL = 0FFH if file not found |
| Notes | The file control block (FCB) is only used with early DOS software and should never be used with new programs. File control blocks do not allow path names, as do the newer file access function codes presented later. Figure A–2 illustrates the structure of the FCB. To open a file, the file must either be present on the disk or be created with function call 16H. |
| **10H** | @CLOSE FILE WITH FCB |
| Entry | AH = 10H<br>DS:DX = address of the opened file control block (FCB) |
| Exit | AL = 00H if file closed<br>AL = 0FFH if error found |
| Notes | Errors that occur usually indicate either that the disk is full or the media is bad. |
| **11H** | @SEARCH FOR FIRST MATCH (FCB) |
| Entry | AH = 11H<br>DS:DX = address of the file control block to be searched |
| Exit | AL = 00H if file found<br>AL = 0FFH if file not found |
| Notes | Wild card characters (? or *) may be used to search for a file name. The ? wild card character matches any character, and the * matches any name or extension. |

| **12H** | @SEARCH FOR NEXT MATCH (FCB) |
|---|---|
| Entry | AH = 12H<br>DS:DX = address of the file control block to be searched |
| Exit | AL = 00H if file found<br>AL = 0FFH if file not found |
| Notes | This function is used after function 11H finds the first matching file name. |
| **13H** | @DELETE FILE USING FCB |
| Entry | AH = 13H<br>DS:DX = address of the file control block to be deleted |
| Exit | AL = 00H if file deleted<br>AL = 0FFH if error occurred |
| Notes | Errors that most often occur are defective media errors. |
| **14H** | @SEQUENTIAL READ (FCB) |
| Entry | AH = 14H<br>DS:DX = address of the file control block to be read |
| Exit | AL = 00H if read successful<br>AL = 01H if end of file reached<br>AL = 02H if DTA had a segment wrap<br>AL = 03H if less than 128 bytes were read |
| **15H** | @SEQUENTIAL WRITE (FCB) |
| Entry | AH = 15H<br>DS:DX = address of the file control block to be written |
| Exit | AL = 00H if write successful<br>AL = 01H if disk is full<br>AL = 02H if DTA had a segment wrap |

| **16H** | @CREATE A FILE (FCB) |
|---|---|
| Entry | AH = 16H<br>DS:DX = address of an unopened file control block |
| Exit | AL = 00H if file created<br>AL = 01H if disk is full |
| **17H** | @RENAME A FILE (FCB) |
| Entry | AH = 17H<br>DS:DX = address of a modified file control block |
| Exit | AL = 00H if file renamed<br>AL = 01H if error occurred |
| Notes | Refer to Figure A–3 for the modified FCB used to rename a file. |
| **18H** | NOT ASSIGNED |
| | |
| **19H** | RETURN CURRENT DRIVE |
| Entry | AH = 19H |
| Exit | AL = current drive |
| Notes | AL = 00H for drive A, 01H for drive B, and so forth. |
| **1AH** | SET DISK TRANSFER AREA |
| Entry | AH = 1AH<br>DS: DX = address of new DTA |
| Notes | The disk transfer area is normally located within the program segment prefix at offset address 80H. The DTA is used by DOS for all disk data transfers using file control blocks. |

| **1BH** | GET DEFAULT DRIVE FILE ALLOCATION TABLE (FAT) |
|---|---|
| Entry | AH = 1BH |
| Exit | AL = number of sectors per cluster<br>DS:BX = address of the media descriptor<br>CX = size of a sector in bytes<br>DX = number of clusters on drive |
| Notes | Refer to Figure A–4 for the format of the media descriptor byte. The DS register is changed by this function, so make sure to save it before using this function. |
| **1CH** | GET ANY DRIVE FILE ALLOCATION TABLE (FAT) |
| Entry | AH = 1CH<br>DL = disk drive number |
| Exit | AL = number of sectors per cluster<br>DS:BX = address of the media descriptor<br>CX = size of a sector in bytes<br>DX = number of clusters on drive |
| **1DH** | NOT ASSIGNED |
| | |
| **1EH** | NOT ASSIGNED |
| | |
| **1FH** | NOT ASSIGNED |
| | |
| **20H** | NOT ASSIGNED |

| **21H** | @RANDOM READ USING FCB |
|---|---|
| Entry | AH = 21H<br>DS:DX = address of opened FCB |
| Exit | AL = 00H if read successful<br>AL = 01H if end of file reached<br>AL = 02H if the segment wrapped<br>AL = 03H if less than 128 bytes read |
| **22H** | @RANDOM WRITE USING FCB |
| Entry | AH = 22H<br>DS:DX = address of opened FCB |
| Exit | AL = 00H if write successful<br>AL = 01H if disk full<br>AL = 02H if the segment wrapped |
| **23H** | @RETURN NUMBER OF RECORDS (FCB) |
| Entry | AH = 23H<br>DS:DX = address of FCB |
| Exit | AL = 00H number of records<br>AL = 0FFH if file not found |
| **24H** | @SET RELATIVE RECORD SIZE (FCB) |
| Entry | AH = 24H<br>DS:DX = address of FCB |
| Notes | Sets the record field to the value contained in the FCB. |

| **25H** | SET INTERRUPT VECTOR |
|---|---|
| Entry | AH = 25H<br>AL = interrupt vector number<br>DS:DX = address of new interrupt procedure |
| Notes | Before changing the interrupt vector, it is suggested that the current interrupt vector is saved using DOS function 35H. This allows a back-link, so the original vector can later be restored. |
| **26H** | CREATE NEW PROGRAM SEGMENT PREFIX |
| Entry | AH = 26H<br>DX = segment address of new PSP |
| Notes | Figure A–5 illustrates the structure of the program segment prefix. |
| **27H** | @RANDOM FILE BLOCK READ (FCB) |
| Entry | AH = 27H<br>CX = the number of records<br>DS:DX = address of opened FCB |
| Exit | AL = 00H if read successful<br>AL = 01H if end of file reached<br>AL = 02H if the segment wrapped<br>AL = 03H if less than 128 bytes read<br>CX = the number of records read |
| **28H** | @RANDOM FILE BLOCK WRITE (FCB) |
| Entry | AH = 28H<br>CX = the number of records<br>DS:DX = address of opened FCB |
| Exit | AL = 00H if write successful<br>AL = 01H if disk full<br>AL = 02H if the segment wrapped<br>CX = the number of records written |

| **29H** | @PARSE COMMAND LINE (FCB) |
|---|---|
| Entry | AH = 29H<br>AL = parse mask<br>DS:SI = address of FCB<br>DS:DI = address of command line |
| Exit | AL = 00H if no file name characters found<br>AL = 01H if file name characters found<br>AL = 0FFH if drive specifier incorrect<br>DS:SI = address of character after name<br>DS:DI = address first byte of FCB |

| **2AH** | READ SYSTEM DATE |
|---|---|
| Entry | AH = 2AH |
| Exit | AL = day of the week<br>CX = the year (1980–2099)<br>DH = the month<br>DL = day of the month |
| Notes | The day of the week is encoded as Sunday = 00H through Saturday = 06H. The year is a binary number equal to 1980 through 2099. |

| **2BH** | SET SYSTEM DATE |
|---|---|
| Entry | AH = 2BH<br>CX = the year (1980–2099)<br>DH = the month<br>DL = day of the month |

| **2CH** | READ SYSTEM TIME |
|---|---|
| Entry | AH = 2CH |
| Exit | CH = hours (0–23)<br>CL = minutes<br>DH = seconds<br>DL = hundredths of seconds |

| **2DH** | SET SYSTEM TIME |
|---|---|
| Entry | AH = 2DH<br>CH = hours<br>CL = minutes<br>DH = seconds<br>DL = hundredths of seconds |
| **2EH** | DISK VERIFY WRITE |
| Entry | AH = 2EH<br>AL = 00H to disable verify on write<br>AL = 01H to enable verify on write |
| **2FH** | READ DISK TRANSFER AREA |
| Entry | AH = 2FH |
| Exit | ES:BX = contains DTA address |
| **30H** | READ DOS VERSION NUMBER |
| Entry | AH = 30H |
| Exit | AH = fractional version number<br>AL = whole number version number |
| Notes | For example, DOS version number 3.2 is returned as a 3 in AL and a 14H in AH. |
| **31H** | TERMINATE AND STAY RESIDENT (TSR) |
| Entry | AH = 31H<br>AL = the DOS return code<br>DX = number of paragraphs to reserve |
| Notes | A paragraph is 16 bytes, and the DOS return code is read at the batch file level with ERRORCODE. |
| **32H** | NOT ASSIGNED |

| **33H** | TEST CONTROL-BREAK |
|---|---|
| Entry | AH = 33H<br>AL = 00H to request current control-break<br>AL = 01H to change control-break<br>DL = 00H to disable control-break<br>DL = 01H to enable control-break |
| Exit | DL = current control-break state |
| **34H** | GET ADDRESS OF InDOS FLAG |
| Entry | AH = 34H |
| Exit | ES:BX = address of InDOS flag |
| Notes | The InDOS flag is available in DOS versions 3.2 or newer and indicates DOS activity. If InDOS = 00H, DOS is inactive or 0FFH if DOS is active. |
| **35H** | READ INTERRUPT VECTOR |
| Entry | AH = 35H<br>AL = interrupt vector number |
| Exit | ES:BX = address stored at vector |
| Notes | This DOS function is used with function 25H to install/remove interrupt handlers. |
| **36H** | DETERMINE FREE DISK SPACE |
| Entry | AH = 36H<br>DL = drive number |
| Exit | AX = FFFFH if drive invalid<br>AX = number of sectors per cluster<br>BX = number of free clusters<br>CX = bytes per sector<br>DX = number of clusters on drive |
| Notes | The default disk drive is DL = 00H, drive A = 01H, drive B = 02H, and so forth. |

| | |
|---|---|
| **37H** | NOT ASSIGNED |
| | |
| **38H** | RETURN COUNTRY CODE |
| Entry | AH = 38H<br>AL = 00H for current country code<br>BX = 16-bit country code<br>DS:DX = data buffer address |
| Exit | AX = error code if carry set<br>BX = counter code<br>DS:DX = data buffer address |
| **39H** | CREATE SUBDIRECTORY |
| Entry | AH = 39H<br>DS:DX = address of ASCII-Z string subdirectory name |
| Exit | AX = error code if carry set |
| Notes | The ASCII-Z string is the name of the subdirectory in ASCII code ended with a 00H instead of a carriage return/line feed. |
| **3AH** | ERASE SUBDIRECTORY |
| Entry | AH = 3AH<br>DS:DX = address of ASCII-Z string subdirectory name |
| Exit | AX = error code if carry set |
| **3BH** | CHANGE SUBDIRECTORY |
| Entry | AH = 3BH<br>DS:DX = address of new ASCII-Z string subdirectory name |
| Exit | AX = error code if carry set |

| **3CH** | CREATE A NEW FILE |
|---|---|
| Entry | AH = 3CH<br>CX = attribute word<br>DS:DX = address of ASCII-Z string file name |
| Exit | AX = error code if carry set<br>AX = file handle if carry cleared |
| Notes | The attribute word can contain any of the following (added together): 01H = read-only access, 02H = hidden file or directory,04H = system file, 08H = volume label, 10H = subdirectory, and 20H = archive bit. In most cases, a file is created with 0000H. |
| **3DH** | OPEN A FILE |
| Entry | AH = 3DH<br>AL = access code<br>DS:DX = address of ASCII-Z string file name |
| Exit | AX = error code if carry set<br>AX = file handle if carry cleared |
| Notes | The access code is AL = 00H for a read-only access, AL = 01H for a write-only access, and AL = 02H for a read/write access. For shared files in a network environment, bit 4 of AL = 1 will deny read/write access, bit 5 of AL = 1 will deny a write access, bits 4 and 5 of AL = 1 will deny read access, bit 6 of AL = 1 denies none, bit 7 of AL = 0 causes the file to be inherited by child, and bit 7 of AL = 1 file is restricted to current process. |
| **3EH** | CLOSE A FILE |
| Entry | AH = 3EH<br>BX = file handle |
| Exit | AX = error code if carry set |

| **3FH** | READ A FILE |
|---|---|
| Entry | AH = 3FH<br>BX = file handle<br>CX = number of bytes to be read<br>DS:DX = address of file buffer to hold data read |
| Exit | AX = error code if carry set<br>AX = number of bytes read if carry cleared |
| **40H** | WRITE A FILE |
| Entry | AH = 40H<br>BX = file handle<br>CX = number of bytes to write<br>DS:DX = address of file buffer that holds write data |
| Exit | AX = error code if carry set<br>AX = number of bytes written if carry cleared |
| **41H** | DELETE A FILE |
| Entry | AH = 41H<br>DS:DX = address of ASCII-Z string file name |
| Exit | AX = error code if carry set |
| **42H** | MOVE FILE POINTER |
| Entry | AH = 42H<br>AL = move technique<br>BX = file handle<br>CX:DX = number of bytes pointer moved |
| Exit | AX = error code if carry set<br>AX:DX = bytes pointer moved |
| Notes | The move technique causes the pointer to move from the start of the file if AL = 00H, from the current location if AL = 01H, and from the end of the file if AL = 02H. The count is stored so DX contains the least-significant 16 bits, and either CX or AX contains the most-significant 16 bits. |

| **43H** | READ/WRITE FILE ATTRIBUTES |
|---|---|
| Entry | AH = 43H<br>AL = 00H to read attributes<br>AL = 01H to write attributes<br>CX = attribute word (see function 3CH)<br>DS:DX = address of ASCII-Z string file name |
| Exit | AX = error code if carry set<br>CX = attribute word of carry cleared |

| **44H** | I/O DEVICE CONTROL (IOTCL) |
|---|---|
| Entry | AH = 44H<br>AL = code (see notes)<br>AL = 01H to write attributes<br>BX = file handle or device number<br>CX = number of bytes<br>DS:DX = data or address |
| Exit | AX = error code if carry set<br>AX and DX = parameters |
| Notes | The codes found in AL are as follows:<br><br>00H = read device status (DX = status)<br>01H = write device status (DX = status written)<br>02H = read data from device (DS:DX = buffer address)<br>03H = write data to device (DS:DX = buffer address)<br>04H = read data from disk drive<br>05H = write data to disk drive<br>06H = read input status (AL = 00H ready or 0FH not ready)<br>07H = read output status (AL = 00H ready or 0FH not ready)<br>08H = removable media? (AL = 00H removable, 01H fixed)<br>09H = local or remote device? (bit 12 of DX set for remote)<br>0AH = local or remote handle? (bit 15 of DX set for remote)<br>0BH = change entry count<br>0CH = generic I/O control for character devices<br>0DH = generic I/O control for block devices<br>0EH = return number of logical devices (AL = number)<br>0FH = change number of logical devices |

| **45H** | DUPLICATE FILE HANDLE |
|---|---|
| Entry | AH = 45H<br>BX = current file handle |
| Exit | AX = error code if carry set<br>AX = duplicate file handle |
| **46H** | FORCE DUPLICATE FILE HANDLE |
| Entry | AH = 46H<br>BX = current file handle<br>CX = new file handle |
| Exit | AX = error code if carry set |
| Notes | This function works like function 45H, except function 45H allows DOS to select the new handle while this function allows the user to select the new handle. |
| **47H** | READ CURRENT DIRECTORY |
| Entry | AH = 47H<br>DL = drive number<br>DS:SI = address of a 64-byte buffer for directory name |
| Exit | DS:SI addresses current directory name if carry cleared |
| **48H** | ALLOCATE MEMORY BLOCK |
| Entry | AH = 48H<br>BX = number of paragraphs to allocate<br>CX = new file handle |
| Exit | BX = largest block available if carry cleared |
| **49H** | RELEASE ALLOCATED MEMORY BLOCK |
| Entry | AH = 49H<br>ES = segment address of block to be released<br>CX = new file handle |
| Exit | Carry indicates an error if set |

| **4AH** | MODIFY ALLOCATED MEMORY BLOCK |
|---|---|
| Entry | AH = 4AH<br>BX = new block size in paragraphs<br>ES = segment address of block to be modified |
| Exit | BX = largest block available if carry cleared |
| **4BH** | LOAD OR EXECUTE A PROGRAM |
| Entry | AH = 4BH<br>AL = function code<br>ES:BX = address of parameter block<br>DS:DX = address of ASCII-Z string command |
| Exit | Carry indicates an error if set |
| Notes | The function codes are: AL = 00H to load and execute a program and AL = 03H to load a program but not execute it. Figure A–6 shows the parameter block used with this function. |
| **4CH** | TERMINATE A PROCESS |
| Entry | AH = 4CH<br>AL = error code |
| Exit | Returns control to DOS |
| Notes | This function returns control to DOS with the error code saved so it can be obtained using DOS ERROR LEVEL batch processing system. We normally use this function with an error code of 00H to return to DOS. |

| **4DH** | READ RETURN CODE |
|---|---|
| Entry | AH = 4DH |
| Exit | AX = return error code |
| Notes | This function is used to obtain the return status code created by executing a program with DOS function 4BH. The return codes are: AX = 0000H for a normal—no error—termination, AX = 0001H for a control-break termination, AX = 0002H for a critical device error, and AX = 0003H for a termination by an INT 31H. |
| **4EH** | FIND FIRST MATCHING FILE |
| Entry | AH = 4EH<br>CX = file attributes<br>DS:DX = address ASCII-Z string file name |
| Exit | Carry is set for file not found |
| Notes | This function searches the current or named directory for the first matching file. Upon exit, the DTA contains the file information. See Figure A–7 for the disk transfer area (DTA). |
| **4FH** | FIND NEXT MATCHING FILE |
| Entry | AH = 4FH |
| Exit | Carry is set for file not found |
| Notes | This function is used after the first file is found with function 4EH. |
| **50H** | SET PROGRAM SEGMENT PREFIX (PSP) ADDRESS |
| Entry | AH = 50H<br>BX = offset address of the new PSP |
| Notes | Extreme care must be used with this function because no error recovery is possible. |

| **51H** | GET PSP ADDRESS |
|---|---|
| Entry | AH = 51H |
| Exit | BX = current PSP segment address |
| **52H** | NOT ASSIGNED |
| | |
| **53H** | NOT ASSIGNED |
| | |
| **54H** | READ DISK VERIFY STATUS |
| Entry | AH = 54H |
| Exit | AL = 00H if verify off<br>AL = 01H if verify on |
| **55H** | NOT ASSIGNED |
| | |
| **56H** | RENAME FILE |
| Entry | AH = 56H<br>ES:DI = address of ASCII-Z string containing new file name<br>DS:DX = address of ASCII-Z string containing file to be renamed |
| Exit | Carry is set for error condition |

| **57H** | READ FILE'S DATE AND TIME STAMP |
|---|---|
| Entry | AH = 57H<br>AL = function code<br>BX = file handle<br>CX = new time<br>DX = new date |
| Exit | Carry is set for error condition<br>CX = time if carry cleared<br>DX = date if carry cleared |
| Notes | AL = 00H to read date and time or 01H to write date and time. |
| **58H** | NOT ASSIGNED |
| | |
| **59H** | GET EXTENDED ERROR INFORMATION |
| Entry | AH = 59H<br>BX = 0000H for DOS version 3.X |
| Exit | AX = extended error code<br>BH = error class<br>BL = recommended action<br>CH = locus |
| Notes | Following are the error codes found in AX:<br><br>0001H = invalid function number<br>0002H = file not found<br>0003H = path not found<br>0004H = no file handles available<br>0005H = access denied<br>0006H = file handle invalid<br>0007H = memory control block failure<br>0008H = insufficient memory<br>0009H = memory block address invalid<br>000AH = environment failure<br>000BH = format invalid<br>000CH = access code invalid<br>000DH = data invalid<br>000EH = unknown unit<br>000FH = disk drive invalid |

| | |
|---|---|
| | 0010H = attempted to remove current directory<br>0011H = not same device<br>0012H = no more files<br>0013H = disk write-protected<br>0014H = unknown unit<br>0015H = drive not ready<br>0016H = unknown command<br>0017H = data error (CRC check error)<br>0018H = bad request structure length<br>0019H = seek error<br>001AH = unknown media type<br>001BH = sector not found<br>001CH = printer out of paper<br>001DH = write fault<br>001EH = read fault<br>001FH = general failure<br>0020H = sharing violation<br>0021H = lock violation<br>0022H = disk change invalid<br>0023H = FCB unavailable<br>0024H = sharing buffer exceeded<br>0025H = code page mismatch<br>0026H = handle end of file operation not completed<br>0027H = disk full<br>0028H–0031H reserved<br>0032H = unsupported network request<br>0033H = remote machine not listed<br>0034H = duplicate name on network<br>0035H = network name not found<br>0036H = network busy<br>0037H = device no longer exists on network<br>0038H = netBIOS command limit exceeded<br>0039H = error in network adapter hardware<br>003AH = incorrect response from network<br>003BH = unexpected network error<br>003CH = remote adapter is incompatible<br>003DH = print queue is full<br>003EH = not enough room for print file<br>003FH = print file was deleted<br>0040H = network name deleted<br>0041H = network access denied<br>0042H = incorrect network device type<br>0043H = network name not found<br>0044H = network name exceeded limit<br>0045H = netBIOS session limit exceeded<br>0046H = temporary pause<br>0047H = network request not accepted |

| | |
|---|---|
| | 0048H = print or disk redirection pause<br>0049H–004FH reserved<br>0050H = file already exists<br>0051H = duplicate FCB<br>0052H = cannot make directory<br>0053H = failure in INT 24H (critical error)<br>0054H = too many redirections<br>0055H = duplicate redirection<br>0056H = invalid password<br>0057H = invalid parameter<br>0058H = network write failure<br>0059H = function not supported by network<br>005AH = required system component not installed<br>0065H = device not selected<br><br>Following are the error class codes found in BH:<br><br>01H = no resources available<br>02H = temporary error<br>03H = authorization error<br>04H = internal software error<br>05H = hardware error<br>06H = system failure<br>07H = application software error<br>08H = item not found<br>09H = invalid format<br>0AH = item blocked<br>0BH = media error<br>0CH = item already exists<br>0DH = unknown error<br><br>Following is the recommended action found in BL:<br><br>01H = retry operation<br>02H = delay and retry operation<br>03H = user retry<br>04H = abort processing<br>05H = immediate exit<br>06H = ignore error<br>07H = retry with user intervention<br><br>Following is a list of locus in CH:<br><br>01H = unknown source<br>02H = block device error<br>03H = network area<br>04H = serial device error<br>05H = memory error |

| **5AH** | CREATE UNIQUE FILE NAME |
|---|---|
| Entry | AH = 5AH<br>CX = attribute code<br>DS:DX = address of the ASCII-Z string directory path |
| Exit | Carry is set for error condition<br>AX = file handle if carry cleared<br>DS:DX = address of the appended directory name |
| Notes | The ASCII-Z file directory path must end with a backslash (\). On exit the directory name is appended with a unique file name. |
| **5BH** | CREATE A DOS FILE |
| Entry | AH = 5BH<br>CX = attribute code<br>DS:DX = address of the ASCII-Z string containing the file name |
| Exit | Carry is set for error condition<br>AX = file handle if carry cleared |
| Notes | The function only works in DOS version 3.X or higher. |
| **5CH** | LOCK/UNLOCK FILE CONTENTS |
| Entry | AH = 5CH<br>BX = file handle<br>CX:DX = offset address of locked/unlocked area<br>SI:DI = number of bytes to lock or unlock beginning at offset |
| Exit | Carry is set for error condition |
| **5DH** | SET EXTENDED ERROR INFORMATION |
| Entry | AH = 5DH<br>AL = 0AH<br>DS:DX = address of the extended error data structure |
| Notes | This function is used by DOS version 3.1 or higher to store extended error information. |

| **5EH** | NETWORK/PRINTER |
|---|---|
| Entry | AH = 5EH<br>AL = 00H (get network name)<br>DS:DX = address of the ASCII-Z string containing network name |
| Exit | Carry is set for error condition<br>CL = netBIOS number if carry cleared |
| Entry | AH = 5EH<br>AL = 02H (define network printer)<br>BX = redirection list<br>CX = length of setup string<br>DS:DX = address of printer setup buffer |
| Exit | Carry is set for error condition |
| Entry | AH = 5EH<br>AL = 03H (read network printer setup string)<br>BX = redirection list<br>DS:DX = address of printer setup buffer |
| Exit | Carry is set for error condition<br>CX = length of setup string if carry cleared<br>ES:DI = address of printer setup buffer |
| **62H** | GET PSP ADDRESS |
| Entry | AH = 62H |
| Exit | BX = segment address of the current program |
| Notes | The function only works in DOS version 3.0 or higher. |

| **65H** | GET EXTENDED COUNTRY INFORMATION |
|---|---|
| Entry | AH = 65H<br>AL = function code<br>ES:DI = address of buffer to receive information |
| Exit | Carry is set for error condition<br>CX = length of country information |
| Notes | The function only works in DOS version 3.3 or higher. |
| **66H** | GET/SET CODE PAGE |
| Entry | AH = 66H<br>AL = function code<br>BX = code page number |
| Exit | Carry is set for error condition<br>BX = active code page number<br>DX = default code page number |
| Notes | A function code in AL of 01H gets the code page number, and a code of 02H sets the code page number. |
| **67H** | SET HANDLE COUNT |
| Entry | AH = 67H<br>BX = number of handles desired |
| Exit | Carry is set for error condition |
| Notes | This function is available for DOS version 3.3 or higher. |
| **68H** | COMMIT FILE |
| Entry | AH = 68H<br>BX = handle number |
| Exit | Carry is set for error condition<br>Else, the date and time stamp is written to directory |
| Notes | This function is available for DOS version 3.3 or higher. |

| **6CH** | EXTENDED OPEN FILE |
|---|---|
| Entry | AH = 6CH<br>AL = 00H<br>BX = open mode<br>CX = attributes<br>DX = open flag<br>DS:SI = address of ASCII-Z string file name |
| Exit | AX = error code if carry is set<br>AX = handle if carry is cleared<br>CX = 0001H file existed and was opened<br>CX = 0002H file did not exist and was created |
| Notes | This function is available for DOS version 4.0 or higher. |

**FIGURE A–2** Contents of the file control block (FCB).

| Offset | Contents |
|---|---|
| 00H | Drive |
| 01H | 8-character filename |
| 09H | 3-character file extension |
| 0CH | Current block number |
| 0EH | Record size |
| 10H | File size |
| 14H | Creation date |
| 16H | Reserved space |
| 20H | Current record number |
| 21H | Relative record number |

**FIGURE A–3** Contents of the modified file control block (FCB).

| Offset | Contents |
|---|---|
| 00H | Drive |
| 01H | 8-character filename |
| 09H | 3-character extension |
| 0CH | Current block number |
| 0EH | Record size |
| 10H | File size |
| 14H | Creation date |
| 16H | Second file name |

**FIGURE A–4** Contents of the media-descriptor byte.

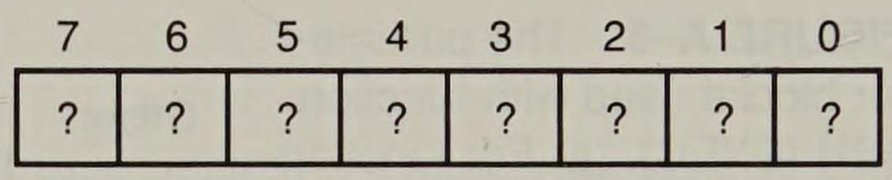



**FIGURE A–5** Contents of the program-segment prefix (PSP).

| Offset | Contents |
|---|---|
| 00H | INT 20H |
| 02H | Top of memory |
| 04H | Reserved |
| 05H | Opcode |
| 06H | Number of bytes in segment |
| 0AH | Terminate address (offset) |
| 0CH | Terminate address (segment) |
| 0EH | Control-break address (offset) |
| 10H | Control-break address (segment) |
| 12H | Critical error address (offset) |
| 14H | Critical error address (segment) |
| 16H | Reserved |
| 2CH | Environment address (segment) |
| 2EH | Reserved |
| 50H | DOS call |
| 52H | Reserved |
| 5CH | File control block 1 |
| 6CH | File control block 2 |
| 80H | Command line length |
| 81H | Command line |

**FIGURE A–6** The parameter blocks used with function 4BH (EXEC). (a) For function code 00H. (b) For function code 03H.

(a)

| Offset | Contents |
|---|---|
| 00H | Environment address (segment) |
| 02H | Command line address (offset) |
| 04H | Command line address (segment) |
| 06H | File control block 1 address (offset) |
| 08H | File control block 1 address (segment) |
| 0AH | File control block 2 address (segment) |
| 0CH | File control block 2 address (offset) |

(b)

| Offset | Contents |
|---|---|
| 00H | Overlay destination segment address |
| 02H | Relocation factor |

**FIGURE A–7** Data transfer area (DTA) used to find a file.

| Offset | Contents |
|---|---|
| 15H | Attributes |
| 16H | Creation time |
| 18H | Creation date |
| 1AH | Low word file size |
| 1CH | High word file size |
| 1EH | Search file name |

## BIOS FUNCTION CALLS

In addition to DOS function call INT 21H, some other BIOS function calls prove useful in controlling the I/O environment of the computer. Unlike INT 21H, which exists in the DOS program, the BIOS function calls are found stored in the BIOS ROM. These BIOS functions directly control the I/O devices with or without DOS loaded into a system.

### INT 10H

The INT 10H BIOS interrupt is often called the video services interrupt because it directly controls the video display in a system. The INT 10H instruction uses register AH to select the video service provided by this interrupt.

## VIDEO MODE SELECTION

The mode of operation for the video display is selected by placing a 00H into AH, followed by one of many mode numbers in AL. Table A–4 lists the modes of operation found in video display systems using standard video modes. The VGA can use any mode listed, while the other displays are more restricted in use. Additional higher-resolution modes are explained later in this section.

Example A–6 lists a short sequence of instructions that place the video display in mode 03H. This mode is available on CGA, EGA, and VGA displays. This mode allows the display to draw test, with 16 colors at various resolutions dependent upon the display adapter.

**EXAMPLE A–6**

```
0000 B4 00                    MOV   AH,0            ;select mode
0002 B0 03                    MOV   AL,3            ;mode is 03H
0004 CD 10                    INT   10H
```

**TABLE A–4** Video display modes

| *Mode* | *Type* | *Columns* | *Rows* | *Resolution* | *Standard* | *Colors* |
|---|---|---|---|---|---|---|
| 00H | Text | 40 | 25 | 320 × 200 | CGA | 2 |
| 00H | Text | 40 | 25 | 320 × 350 | EGA | 2 |
| 00H | Text | 40 | 25 | 360 × 400 | VGA | 2 |
| 01H | Text | 40 | 25 | 320 × 200 | CGA | 16 |
| 01H | Text | 40 | 25 | 320 × 350 | EGA | 16 |
| 01H | Text | 40 | 25 | 360 × 400 | VGA | 16 |
| 02H | Text | 80 | 25 | 640 × 200 | CGA | 2 |
| 02H | Text | 80 | 25 | 640 × 350 | EGA | 2 |
| 02H | Text | 80 | 25 | 720 × 400 | VGA | 2 |
| 03H | Text | 80 | 25 | 640 × 200 | CGA | 16 |
| 03H | Text | 80 | 25 | 640 × 350 | EGA | 16 |
| 03H | Text | 80 | 25 | 720 × 400 | VGA | 16 |
| 04H | Graphics | 40 | 25 | 320 × 200 | CGA | 4 |
| 05H | Graphics | 40 | 25 | 320 × 200 | CGA | 2 |
| 06H | Graphics | 80 | 25 | 640 × 200 | CGA | 2 |
| 07H | Text | 80 | 25 | 720 × 350 | EGA | 4 |
| 07H | Text | 80 | 25 | 720 × 400 | VGA | 4 |
| 0DH | Graphics | 80 | 25 | 320 × 200 | CGA | 16 |
| 0EH | Graphics | 80 | 25 | 640 × 200 | CGA | 16 |
| 0FH | Graphics | 80 | 25 | 640 × 350 | EGA | 4 |
| 10H | Graphics | 80 | 25 | 640 × 350 | EGA | 16 |
| 11H | Graphics | 80 | 30 | 640 × 480 | VGA | 2 |
| 12H | Graphics | 80 | 30 | 640 × 480 | VGA | 16 |
| 13H | Graphics | 40 | 25 | 320 × 200 | VGA | 256 |

## CURSOR CONTROL

Table A–5 shows the function codes used to control the cursor on the video display. These cursor control functions will work on any video display from the CGA display to the latest super VGA display.

If an SVGA (super VGA), EVGA (extended VGA), or XVGA (also extended VGA) adapter is available, the super VGA mode is set by using INT 10H function call AX = 4F02H, with BX = to the VGA mode for these advanced display adapters. This conforms to the VESA standard for VGA adapters. Table A–6 shows the modes selected by register BX for this INT 10H function call. Most video cards are equipped with a driver called VVESA.COM, or VVESA.SYS, which conforms the card to the VESA standard functions.

## INT 11H

This function is used to determine the type of equipment installed in the system. To use this call, the AX register is loaded with an FFFFH, then the INT 11H instruction is executed. In return, an INT 11H provides information as listed in Figure A–8.

## INT 12H

The memory size is returned by the INT 12H instruction. After executing the INT 12H instruction, the AX register contains the number of 1K byte blocks of memory (conventional memory in the first 1M byte of address space) installed in the computer.

## INT 13H

This call controls the floppy disks and the fixed or hard disk drives attached to the system. Table A–7 lists the functions available to this interrupt via register AH. The direct control of a floppy disk or hard disk can lead to problems. Therefore, we only provide a listing of the functions without detail on their usage. Before using these functions, refer to the BIOS literature available from the company that produced your version of the BIOS ROM.

## INT 14H

Interrupt 14H controls the serial COM (communications) ports attached to the computer. The computer system contains two COM ports, COM1 and COM2, unless you have a newer AT-style machine where the number of communications ports are extended to COM3 and COM4. Communications ports are normally controlled with software packages that allow data transfer through a modem and telephone lines. The INT 14H instruction controls these ports as illustrated in Table A–8.

## INT 15H

The INT 15H instruction controls many of the various I/O devices interfaced to the computer. It also allows access to protected-mode operation and the extended memory system on an 80286, 80386 or 80486 system. Table A–9 lists the functions supported by INT 15H.

**TABLE A–5** Video BIOS (INT 10H) functions (pp. 675–678)

| **00H** | SELECT VIDEO MODE |
|---|---|
| Entry | AH = 00H<br>AL = mode number |
| Exit | Mode changed and screen cleared |
| **01H** | SELECT CURSOR TYPE |
| Entry | AH = 01H<br>CH = starting line number<br>CL = ending line number |
| Exit | Cursor size changed |
| **02H** | SELECT CURSOR POSITION |
| Entry | AH = 02H<br>BH = page number (usually 0)<br>DH = row number (beginning with 0)<br>DL = column number (beginning with 0) |
| Exit | Changes cursor to new position |
| **03H** | READ CURSOR POSITION |
| Entry | AH = 03H<br>BH = page number |
| Exit | CH = starting line (cursor size)<br>CL = ending line (cursor size)<br>DH = current row<br>DL = current column |
| **04H** | READ LIGHT PEN |
| Entry | AH = 04H (not supported in VGA) |
| Exit | AH = 0, light pen triggered<br>BX = pixel column<br>CX = pixel row<br>DH = character row<br>DL = character column |

| **05H** | SELECT DISPLAY PAGE |
|---|---|
| Entry | AH = 05H<br>AL = page number |
| Exit | Page number selected. Following are the valid page numbers.<br><br>Modes 0 and 1 support pages 0–7<br>Modes 2 and 3 support pages 0–7<br>Modes 4, 5, and 6 support page 0<br>Modes 7 and D support pages 0–7<br>Mode E supports pages 0–3<br>Modes F and 10 support pages 0–1<br>Modes 11, 12, and 13 support page 0 |
| **06H** | SCROLL PAGE UP |
| Entry | AH = 06H<br>AL = number of lines to scroll (0 clears window)<br>BH = character attribute for new lines<br>CH = top row of scroll window<br>CL = left column of scroll window<br>DH = bottom row of scroll window<br>DL = right column of scroll window |
| Exit | Scrolls window from the bottom toward the top of the screen.<br>Blank lines fill the bottom using the character attribute in BH. |
| **07H** | SCROLL PAGE DOWN |
| Entry | AH = 07H<br>AL = number of lines to scroll (0 clears window)<br>BH = character attribute for new lines<br>CH = top row of scroll window<br>CL = left column of scroll window<br>DH = bottom row of scroll window<br>DL = right column of scroll window |
| Exit | Scrolls window from the top toward the bottom of the screen.<br>Blank lines fill from the top using the character attribute in BH. |

| **08H** | READ ATTRIBUTE/CHARACTER AT CURRENT CURSOR POSITION |
|---|---|
| Entry | AH = 08H<br>BH = page number |
| Exit | AL = ASCII character code<br>AH = character attribute<br>Note: this function does not advance the cursor. |
| **09H** | WRITE ATTRIBUTE/CHARACTER AT CURRENT CURSOR POSITION |
| Entry | AH = 09H<br>AL = ASCII character code<br>BH = page number<br>BL = character attribute<br>CX = number of characters to write |
| Exit | Note: this function does not advance the cursor. |
| **0AH** | WRITE CHARACTER AT CURRENT CURSOR POSITION |
| Entry | AH = 0AH<br>AL = ASCII character code<br>BH = page number<br>CX = number of characters to write |
| Exit | Note: this function does not advance the cursor. |
| **0FH** | READ VIDEO MODE |
| Entry | AH = 0FH |
| Exit | AL = current video mode<br>AH = number of character columns<br>BH = page number |

| **10H** | SET VGA PALETTE REGISTER |
|---|---|
| Entry | AH = 10H<br>AL = 10H<br>BX = color number (0–255)<br>CH = green (0–63)<br>CL = blue (0–63)<br>DH = red (0–63) |
| Exit | Palette register color is changed. Note: the first 16 colors (0–15) are used in the 16-color, VGA text mode and other modes. |
| **10H** | READ VGA PALETTE REGISTER |
| Entry | AH = 10H<br>AL = 15H<br>BX = color number (0–255) |
| Exit | CH = green<br>CL = blue<br>DH = red |
| **11H** | GET ROM CHARACTER SET |
| Entry | AH = 11H<br>AL = 30H<br>BH = 2 = ROM 8 × 14 character set<br>BH = 3 = ROM 8 × 8 character set<br>BH = 4 = ROM 8 × 8 extended character set<br>BH = 5 = ROM 9 × 14 character set<br>BH = 6 = ROM 8 × 16 character set<br>BH = 7 = ROM 9 × 16 character set |
| Exit | CX = bytes per character<br>DL = rows per character<br>ES:BP = address of character set |

## INT 16H

The INT 16H instruction is used as a keyboard interrupt. This interrupt is accessed by DOS interrupt INT 21H, but can be accessed directly. Table A–10 shows the functions performed by INT 16H.

**TABLE A–6** Extended VGA functions

| *BX* | *Function* |
|---|---|
| 100H | 640 × 400 with 256 colors |
| 101H | 640 × 480 with 256 colors |
| 102H | 800 × 600 with 16 colors |
| 103H | 800 × 600 with 256 colors |
| 104H | 1,024 × 768 with 16 colors |
| 105H | 1,024 × 768 with 256 colors |
| 106H | 1,280 × 1,024 with 16 colors |
| 107H | 1,280 × 1,024 with 256 colors |
| 108H | 80 × 60 in text mode |
| 109H | 132 × 25 in text mode |
| 10AH | 132 × 43 in text mode |
| 10BH | 132 × 50 in text mode |
| 10CH | 132 × 60 in text mode |

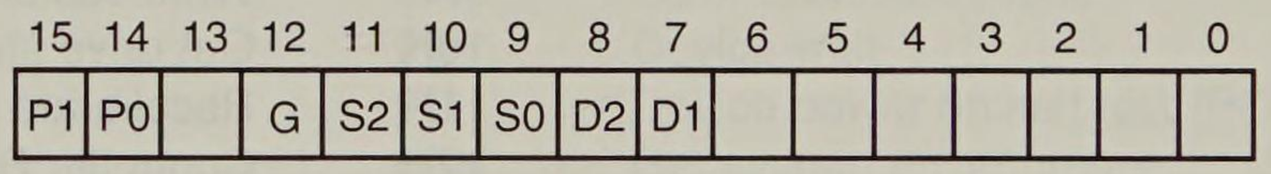



**FIGURE A–8** The contents of AX as it indicates the equipment attached to the computer.

## INT 17H

The INT 17H instruction accesses the parallel printer port, labeled LPT1 in most systems. Table A–11 lists the three functions available for the INT 17H instruction.

## DOS SYSTEM MEMORY MAP

Figure A–9 illustrates the memory map used by a DOS computer system. The first 1M byte of memory is listed with all the areas containing different devices and programs used with DOS. The transient program area (TPA) is where DOS applications programs are loaded and executed. The size of the TPA is usually slightly over 500K byte, unless many TSR programs and drivers fill memory before the TPA.

## DOS LOW-MEMORY ASSIGNMENTS

Table A–12 shows the low-memory assignments (00000H–005FFH) for the DOS-based microprocessor system. This area of memory contains the interrupt vectors, BIOS data area, and the DOS/BIOS data area illustrated in Figure A–9.

**TABLE A–7** Disk I/O function via INT 13H

| *AH* | *Function* |
|---|---|
| 00H | Reset disk system |
| 01H | Get disk system status into AL |
| 02H | Read sector |
| 03H | Write sector |
| 04H | Verify sector |
| 05H | Format track |
| 06H | Format bad track |
| 07H | Format drive |
| 08H | Get drive parameters |
| 09H | Initialize fixed disk characteristics |
| 0AH | Read long sector |
| 0BH | Write long sector |
| 0CH | Seek |
| 0DH | Reset fixed disk system |
| 0EH | Read sector buffer |
| 0FH | Write sector buffer |
| 10H | Get drive status |
| 11H | Recalibrate drive |
| 12H | Controller RAM diagnostics |
| 13H | Controller drive diagnostics |
| 14H | Controller internal diagnostics |
| 15H | Get disk type |
| 16H | Get disk change status |
| 17H | Set disk type |
| 18H | Set media type |
| 19H | Park heads |
| 1AH | Format ESDI drive |

**TABLE A–8** COM port interrupt INT 14H

| *AH* | *Function* |
|---|---|
| 00H | Initialize communications port |
| 01H | Send character |
| 02H | Receive character |
| 03H | Get COM port status |
| 04H | Extended initialize communications port |
| 05H | Extended communications port control |

**TABLE A–9** The I/O subsystem interrupt INT 15H

| *AH* | *Function* |
|---|---|
| 00H | Cassette motor on |
| 01H | Cassette motor off |
| 02H | Read cassette |
| 03H | Write cassette |
| 0FH | Format ESDI drive periodic interrupt |
| 21H | Keyboard intercept |
| 80H | Device open |
| 81H | Device closed |
| 82H | Process termination |
| 83H | Event wait |
| 84H | Read joystick |
| 85H | System request key |
| 86H | Delay |
| 87H | Move extended block of memory |
| 88H | Get extended memory size |
| 89H | Enter protected mode |
| 90H | Device wait |
| 91H | Device power on self test (POST) |
| C0H | Get system environment |
| C1H | Get address of extended BIOS data area |
| C2H | Mouse pointer |
| C3H | Set watch-dog timer |
| C4H | Programmable option select |

**TABLE A–10** Keyboard interrupt INT 16H

| *AH* | *Function* |
|---|---|
| 00H | Read keyboard character |
| 01H | Get keyboard status |
| 02H | Get keyboard flags |
| 03H | Set repeat rate |
| 04H | Set keyclick |
| 05H | Push character and scan code |

**TABLE A–11** Parallel printer interrupt INT 17H

| *AH* | *Function* |
|---|---|
| 00H | Print character |
| 01H | Initialize printer |
| 02H | Get printer status |

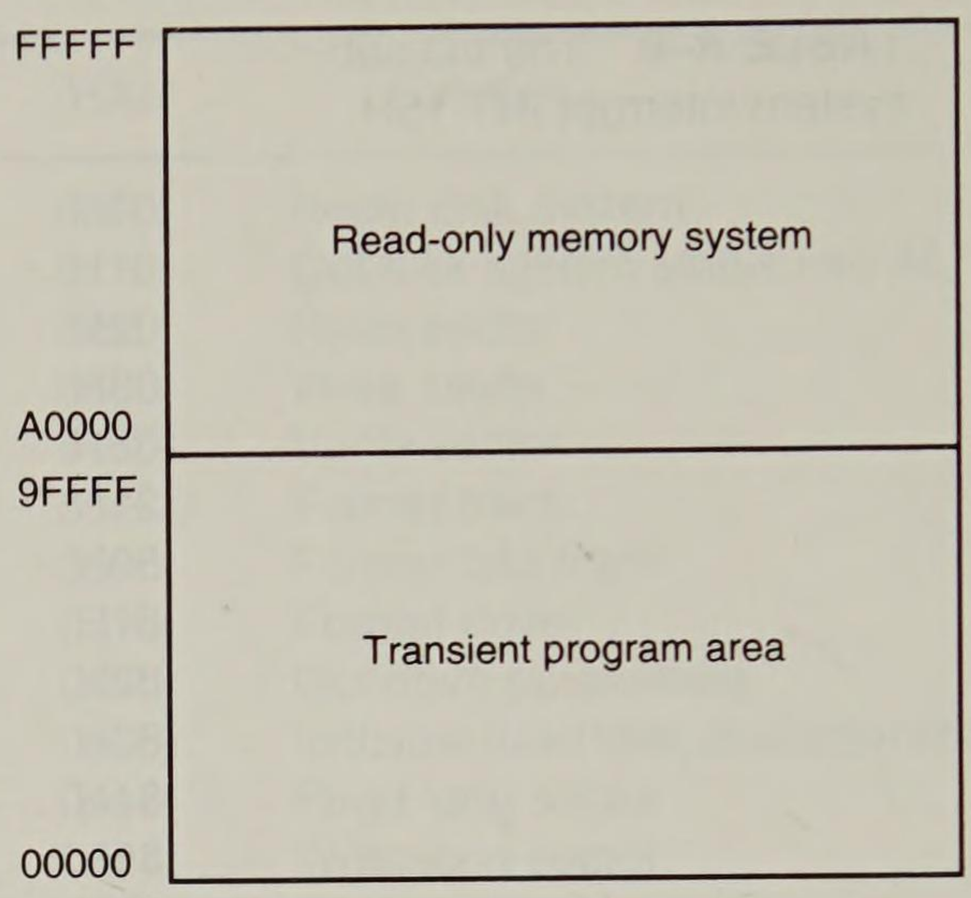



**FIGURE A–9** Memory map of DOS illustrating the first 1M byte of memory.

**TABLE A–12** DOS low-memory assignments

| *Location* | *Purpose* | |
|---|---|---|
| 00000H–002FFH | System interrupt vectors | |
| 00300H–003FFH | System interrupt vectors, power on, and bootstrap area | |
| 00400H–00407H | COM1–COM4 I/O port base addresses | |
| 00408H–0040FH | LPT1–LPT4 I/O port base addresses | |
| 00410H–00411H | Equipment flag word, returned in AX by an INT 11H | |
| | Bits 15–14 | Number of parallel printers (LPT1–LPT4) |
| | Bit 13 | Internal MODEM installed |
| | Bit 12 | Joystick installed |
| | Bits 11–9 | Number of serial ports (COM1–COM4) |
| | Bit 8 | Unused |
| | Bits 7–6 | Number of disk drives |
| | Bits 5–4 | Video mode |
| | Bits 3–2 | Unused |
| | Bit 1 | Math coprocessor installed |
| | Bit 0 | Disk installed |
| 00412H | Reserved | |
| 00413H–00414H | Memory size in K bytes (0–640K) | |
| 00415H–00416H | Reserved | |
| 00417H | Keyboard control byte | |
| | Bit 7 | Insert locked |
| | Bit 6 | Caps locked |
| | Bit 5 | Numbers locked |
| | Bit 4 | Scroll locked |
| | Bit 3 | Alternate key pressed |
| | Bit 2 | Control key pressed |
| | Bit 1 | Left shift key pressed |
| | Bit 0 | Right shift key pressed |
| 00418H | Keyboard control byte | |

**TABLE A–12** (Continued)

| *Location* | *Purpose* | |
|---|---|---|
| | Bit 7 | Insert key pressed |
| | Bit 6 | Caps lock key pressed |
| | Bit 5 | Numbers lock key pressed |
| | Bit 4 | Scroll lock key pressed |
| | Bit 3 | Pause locked |
| | Bit 2 | System request key pressed |
| | Bit 1 | Left alternate key pressed |
| | Bit 0 | Right control key pressed |
| 00419H | Alternate keyboard entry | |
| 0041AH–0041BH | Keyboard buffer header pointer | |
| 0041CH–0041DH | Keyboard buffer tail pointer | |
| 0041EH–0043DH | 32-byte keyboard buffer area | |
| 0043EH–00448H | Disk drive control area | |
| 00449H–00466H | Video control data area | |
| 00467H–0046BH | Reserved | |
| 0046CH–0046FH | Timer counter | |
| 00470H | Timer overflow | |
| 00471H | Break key state | |
| 00472H–00473H | Reset flag | |
| 00474H–00477H | Hard disk drive data area | |
| 00478H–0047BH | LPT1–LPT4 timeout area | |
| 0047CH–0047FH | COM1–COM4 timeout area | |
| 00480H–00481H | Keyboard buffer start offset pointer | |
| 00482H–00483H | Keyboard buffer end offset pointer | |
| 00484H–0048AH | Video control data area | |
| 0048BH–00495H | Hard drive control area | |
| 00496H | Keyboard mode, state, and type flag | |
| 00497H | Keyboard LED flags | |
| 00498H–00499H | Offset address of user wait complete flag | |
| 0049AH–0049BH | Segment address of user wait complete flag | |
| 0049CH–0049DH | User wait count (low word) | |
| 0049EH–0049FH | User wait count (high word) | |
| 004A0H | Wait active flag | |
| 004A1H–004A7H | Reserved | |
| 004A8H–004ABH | Pointer to video parameters | |
| 004ACH–004EFH | Reserved | |
| 004F0H–004FFH | Applications program communications area | |
| 00500H | Print screen status | |
| 00501H–00503H | Reserved | |
| 00504H | Single drive mode status | |
| 00505H–0050FH | Reserved | |
| 00510H–00521H | Used by ROM BASIC | |
| 00522H–0052FH | Used by DOS for disk initialization | |
| 00530H–00533H | Used by MODE command | |
| 00534H–005FFH | Reserved | |

## DOS VERSION 5/6 MEMORY MAP

Microsoft DOS version 5/6 has a slightly different memory map than earlier versions of DOS because of its ability to load drivers and programs in the system area. If the microprocessor is an 80386 or 80486, ROM memory located between addresses A0000H and FFFFFH can be backfilled with extended memory for drivers and programs. In many systems, memory area D0000H–DFFFFH is unused, as is E0000H–EFFFFH. These areas can be filled with extended memory through memory paging found in the 80386 and 80486 microprocessors. This new memory area can then be filled with and addressed by normal real-mode memory programs, extending the memory available to DOS applications.

The drivers HIMEM.SYS and EMM386.SYS are used to accomplish the backfilling. If you want to use memory area E0000H–EFFFFH, you must load EMM386.SYS as EMM386.SYS I=E000–EFFF. Using these drivers increases the DOS TPA to more than 600K bytes. A typical CONFIG.SYS file for DOS 5/6 appears in Example A–7. Notice that drivers after EMM386.SYS are loaded in high memory with the DEVICEHIGH directive instead of the DEVICE directive. Programs are loaded using the LOADHIGH or LH directive in front of the program name.

**EXAMPLE A–7**

```
(CONFIG.SYS file)

FILES=30
BUFFERS=30
STACKS=64,128
FCBS=48
SHELL=C:\DOS\COMMAND.COM C:\DOS\ /E:256 /P
DEVICE=C:\DOS\HIMEM.SYS
DOS=HIGH,UMB
DEVICE=C:\DOS\EMM386.EXE I=C800-EFFF NOEMS
DEVICEHIGH SIZE=1EB0 C:\LASERLIB\SONY_CDU.SYS /D:SONY_001 /B:340 /Q:* /T:* /M:H
DEVICEHIGH SIZE=0190 C:\DOS\SETVER.EXE
DEVICEHIGH SIZE=3150 C:\MOUSE1\MOUSE.SYS
LASTDRIVE = F


(AUTOEXEC.BAT file)

PATH C:\DOS;C:\;C:\MASM\BIN;C:\MASM\BINB\;C:\UTILITY;C:\WS;C:\LASERLIB
SET BLASTER=A220 I7 D1 T3
SET INCLUDE=C:\MASM\INCLUDE\
SET HELPFILES=C:\MASM\HELP\*.HLP
SET INIT=C:\MASM\INIT\
SET ASMEX=C:\MASM\SAMPLES\
SET TMP=C:\MASM\TMP
SET SOUND=C:\SB
LOADHIGH C:\LASERLIB\MSCDEX.EXE /D:SONY_001 /L:F /M:8
LOADHIGH C:\LASERLIB\LLTSR.EXE ALT-Q
LOADHIGH C:\DOS\FASTOPEN C:=256
LOADHIGH C:\DOS\DOSKEY /BUFSIZE=1024
LOADHIGH C:\LASERLIB\PRINTF.COM
DOSKEY GO=DOSSHELL
DOSSHELL
```

# APPENDIX B

# Instruction Set Summary

The instruction set summary, which follows this introduction, contains a complete listing of all instructions for the 8086, 8088, 80286, 80386, 80486, and Pentium microprocessors. Numeric coprocessor instructions for the 8087 through the Pentium appear in Chapter 8.

Each instruction entry lists the mnemonic opcode and a brief description of the purpose of the instruction. Also listed is the binary machine language coding for each instruction and any other data required to form the instruction, such as displacement or immediate data. Next to the binary machine language version of the instruction are the flag register bits and any changes that might occur for a given instruction. In this listing a blank indicates no change, a ? indicates a change with an unpredictable outcome, a * indicates a predictable change, a 1 indicates the flag is set, and a 0 indicates the flag is cleared.

Before the instruction listing begins, some information about the bit settings in the binary machine language versions of the instructions is required. Table B–1 shows the modifier bits, coded as oo in the instruction listings, so instructions can be formed with a register, displacement, or no displacement.

Table B–2 lists the memory-addressing modes available with the register/memory field, coded as mmm. This table applies to all versions of the microprocessor.

Table B–3 lists the register options (rrr) when encoded for either an 8-bit or a 16-bit register. This table also lists the 32-bit registers used with the 80386, 80486, and Pentium microprocessors.

**TABLE B–1** The modifier bits, coded as oo in the instruction listing

| *oo* | *Function* |
|---|---|
| 00 | If mmm = 110, then a displacement follows the opcode; otherwise, no displacement is used |
| 01 | An 8-bit signed displacement follows the opcode |
| 10 | A 16-bit signed displacement follows the opcode |
| 11 | mmm specifies a register, instead of an addressing mode |

**TABLE B–2** Register/memory field (mmm) description

| *mmm* | *Function* |
|---|---|
| 000 | DS:[BX+SI] |
| 001 | DS:[BX+DI] |
| 010 | SS:[BP+SI] |
| 011 | SS:[BP+DI] |
| 100 | DS:[SI] |
| 101 | DS:[DI] |
| 110 | SS:[BP] |
| 111 | DS:[BX] |

**TABLE B-3** Register field (rrr) options

| *rrr* | *W = 0* | *W = 1* | *32-bit Register* |
|---|---|---|---|
| 000 | AL | AX | EAX |
| 001 | CL | CX | ECX |
| 010 | DL | DX | EDX |
| 011 | BL | BX | EBX |
| 100 | AH | SP | ESP |
| 101 | CH | BP | EBP |
| 110 | DH | SI | ESI |
| 111 | BH | DI | EDI |

Table B–4 lists the segment register bit assignments (rrr) for the MOV, PUSH, and POP instructions, which use these segment registers.

When the 80386, 80486, and Pentium microprocessors are used, some of the definitions provided in the prior tables will change. Refer to Tables B–5 and B–6 for these changes as they apply to the 80386, 80486, and Pentium microprocessors.

The instruction set summary that follows lists all of the instructions, with examples, for the 8086, 8088, 80286, 80386, 80486, and Pentium microprocessors. Missing are the segment override prefixes, CS (2EH), SS (36H), DS (3EH), ES (26H), FS (64H), and GS (65H). These prefixes are one byte in length and placed in memory before the instruction that is prefixed.

Table B–7 lists the effective address calculations that apply only to the 8086 and 8088 microprocessors as ea or a in the instruction set summary. For example, the 8086 ADD DATA,AL instruction requires 16 + ea clocks. Because the table lists displacement addressing as requiring 6 clocks, this instruction requires 22 clocks to execute. Note that all times listed are maximum times and that, in some cases, the microprocessor may execute the instruction in less time.

The D bit, in the code-segment descriptor, indicates the default size of the operand and the addresses for the 80386, 80486, and Pentium microprocessors. If D = 1, then all addresses and operands are 32 bits; if D = 0, all addresses and operands are 16 bits. In the real mode, the D bit is set to zero by the 80386, 80486, and Pentium microprocessors so operands and addresses are 16 bits.

**TABLE B–4** Register field assignments (rrr) that are used to represent the segment registers

| *rrr* | *Register* |
|---|---|
| 000 | ES |
| 001 | CS |
| 010 | SS |
| 011 | DS |

**TABLE B–5** Index registers specified with rrr in the 80386, 80486, and Pentium microprocessors

| *rrr* | *Index Register* |
|---|---|
| 000 | EAX |
| 001 | ECX |
| 010 | EDX |
| 011 | EBX |
| 100 | No index |
| 101 | EBP |
| 110 | ESI |
| 111 | EDI |

**TABLE B–6** Possible combinations of oo, mmm, and rrr for the 80386, 80486, and Pentium instruction set using the 32-bit addressing mode

| *oo* | *mmm* | *rrr* | *Function* |
|---|---|---|---|
| 00 | 000 | — | DS:[EAX] |
| 00 | 001 | — | DS:[ECX] |
| 00 | 010 | — | DS:[EDX] |
| 00 | 011 | — | DS:[EBX] |
| 00 | 100 | 000 | DS:[EAX+scaled-index] |
| 00 | 100 | 001 | DS:[ECX+scaled-index] |
| 00 | 100 | 010 | DS:[EDX+scaled-index] |
| 00 | 100 | 011 | DS:[EBX+scaled-index] |
| 00 | 100 | 100 | SS:[ESP+scaled-index] |
| 00 | 100 | 101 | DS:[disp32+scaled-index] |
| 00 | 100 | 110 | DS:[ESI+scaled-index] |
| 00 | 100 | 111 | DS:[EDI+scaled-index] |
| 00 | 101 | — | DS:disp32 |
| 00 | 110 | — | DS:[ESI] |
| 00 | 111 | — | DS:[EDI] |
| 01 | 000 | — | DS:[EAX+disp8] |
| 01 | 001 | — | DS:[ECX+disp8] |
| 01 | 010 | — | DS:[EDX+disp8] |
| 01 | 011 | — | DS:[EBX+disp8] |
| 01 | 100 | 000 | DS:[EAX+scaled-index+disp8] |
| 01 | 100 | 001 | DS:[ECX+scaled-index+disp8] |
| 01 | 100 | 010 | DS:[EDX+scaled-index+disp8] |
| 01 | 100 | 011 | DS:[EBX+scaled-index+disp8] |
| 01 | 100 | 100 | SS:[ESP+scaled-index+disp8] |
| 01 | 100 | 101 | SS:[EBP+scaled-index+disp8] |

**TABLE B–6** (Continued)

| | | | |
|---|---|---|---|
| 01 | 100 | 110 | DS:[ESI+scaled-index+disp8] |
| 01 | 100 | 111 | DS:[EDI+scaled-index+disp8] |
| 01 | 101 | — | SS:[EBP+disp8] |
| 01 | 110 | — | DS:[ESI+disp8] |
| 01 | 111 | — | DS:[EDI+disp8] |
| 10 | 000 | — | DS:[EAX+disp32] |
| 10 | 001 | — | DS:[ECX+disp32] |
| 10 | 010 | — | DS:[EDX+disp32] |
| 10 | 011 | — | DS:[EBX+disp32] |
| 10 | 100 | 000 | DS:[EAX+scaled-index+disp32] |
| 10 | 100 | 001 | DS:[ECX+scaled-index+disp32] |
| 10 | 100 | 010 | DS:[EDX+scaled-index+disp32] |
| 10 | 100 | 011 | DS:[EBX+scaled-index+disp32] |
| 10 | 100 | 100 | SS:[ESP+scaled-index+disp32] |
| 10 | 100 | 101 | SS:[EBP+scaled-index+disp32] |
| 10 | 100 | 110 | DS:[ESI+scaled-index+disp32] |
| 10 | 100 | 111 | DS:[EDI+scaled-index+disp32] |
| 01 | 101 | — | SS:[EBP+disp32] |
| 01 | 110 | — | DS:[ESI+disp32] |
| 01 | 111 | — | DS:[EDI+disp32] |

*Note:* disp8 = 8-bit displacement. disp32 = 32-bit displacement.

**TABLE B–7** Effective address calculations for the 8086 and 8088

| *Type* | *Clocks* | *Example* |
|---|---|---|
| Base or index | 5 | MOV CL,[DI] |
| Displacement | 3 | MOV AL,DATA |
| Base plus index [BP+DI] or [BX+SI] | 7 | MOV BL,[BP+DI] |
| Base plus index [BP+SI] or [BX+DI] | 8 | MOV CL,[BP+SI] |
| Displacement plus base or index | 9 | MOV DH,[DI+20H] |
| Base plus index plus displacement [BP+DI+disp] or [BX+SI+disp] | 11 | MOV CX,DATA[BX+SI] |
| Base plus index plus displacement [BP+SI+disp] or [BX+DI+disp] | 12 | MOV CX,[BX+DI+2] |
| Segment override prefix | ea+2 | MOV AL,ES:DATA |

The address-size prefix (67H) must be placed before instructions in the 80386, 80486, and Pentium to change the default size as selected by the D bit. For example, the MOV AX,[ECX] instruction must have the address-size prefix placed before it in machine code if the default size is 16 bits. If the default size is 32 bits, the address prefix is not needed with this instruction. The operand-override prefix (66H) functions in much the same way as the address-size prefix. If the operand size is 16 bits, and the D bit selects 32-bit operands and addresses, this instruction requires the operand-size prefix.

The Pentium microprocessor can execute two instructions that are not dependent on each other simultaneously. This means, for example, that the ADD AL,6 instruction, listed as 1 clock, may actually execute with another instruction listed as 1 clock. As a result, two instructions are executed in one clocking period.

## INSTRUCTION SET SUMMARY (pp. 689–770)

| **AAA** ASCII adjust AL after addition | | |
|---|---|---|
| 00110111<br>Example | O D I T S Z A P C<br>? ? ? * ? *<br> | <br>Clocks |
| AAA | 8086 | 8 |
| | 8088 | 8 |
| | 80286 | 3 |
| | 80386 | 4 |
| | 80486 | 3 |
| | Pentium | 3 |

| **AAD** ASCII adjust AX before division | | |
|---|---|---|
| 11010101 00001010<br>Example | O D I T S Z A P C<br>? * * ? * ?<br> | <br>Clocks |
| AAD | 8086 | 60 |
| | 8088 | 60 |
| | 80286 | 14 |
| | 80386 | 19 |
| | 80486 | 14 |
| | Pentium | 10 |

## AAM ASCII adjust AX after multiplication

11010100 00001010

O D I T S Z A P C
? ? * * ? * ?

Example

| | Clocks |
|---|---|
| AAM | |
| 8086 | 83 |
| 8088 | 83 |
| 80286 | 16 |
| 80386 | 17 |
| 80486 | 15 |
| Pentium | 18 |

## AAS ASCII adjust AL after subtraction

00111111

O D I T S Z A P C
? ? ? * ? * *

Example

| | Clocks |
|---|---|
| AAS | |
| 8086 | 8 |
| 8088 | 8 |
| 80286 | 3 |
| 80386 | 4 |
| 80486 | 3 |
| Pentium | 3 |

## ADC Addition with carry

000100dw oorrrmmm disp

O D I T S Z A P C
* * * * * * *

Format

| Examples | | Clocks |
|---|---|---|
| ADC reg,reg | ADC AX,BX<br>ADC AL,BL<br>ADC EAX,EBX<br>ADC CX,SI<br>ADC ESI,EDI | |
| | 8086 | 3 |
| | 8088 | 3 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 1 |
| | Pentium | 1/2 |

| | | | |
|---|---|---|---|
| ADC mem,reg | ADC DATAY,AL<br>ADC LIST,SI<br>ADC DATAX [DI],CL<br>ADC [EAX],BL<br>ADC [EBX+2*ECX],EDX | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |
| ADC reg,mem | ADC BL,DATA1<br>ADC SI,LIST<br>ADC CL,DATA2 [DI]<br>ADC CL,[EAX]<br>ADC EDX,[EBX+100H] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |
| 100000sw oo010mmm disp data<br>Format | Examples | | Clocks |
| ADC reg,imm | ADC CX,3<br>ADC DI,1AH<br>ADC DL,34H<br>ADC EAX,12345<br>ADC CX,1234H | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| ADC mem,imm | ADC DATA4,33<br>ADC LIST,'A'<br>ADC DATA [DI],2<br>ADC BYTE PTR [EAX],3<br>ADC WORD PTR [DI],669H | 8086 | 17 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 0001010w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| ADC acc,imm | ADC AX,3<br>ADC AL,1AH<br>ADC AH,34<br>ADC EAX,3<br>ADC AL,'Z' | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## ADD Addition

000000dw oorrrmmm disp

O D I T S Z A P C
* * * * * * *

| Format | Examples | | Clocks |
|---|---|---|---|
| ADD reg,reg | ADD AX,BX<br>ADD AL,BL<br>ADD EAX,EBX<br>ADD CX,SI<br>ADD ESI,EDI | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| ADD mem,reg | ADD DATA5,AL<br>ADD LIST,SI<br>ADD DATA6 [DI],CL<br>ADD [EAX],CL<br>ADD [EBX+4*EDX],EBX | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |
| ADD reg,mem | ADD BL,DATA<br>ADD SI,LIST<br>ADD CL,DATA7 [DI]<br>ADD CL,[EAX]<br>ADD EDX,[EBX+200H] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

| 100000sw oo000mmm disp data<br>Format | Examples | | Clocks |
|---|---|---|---|
| ADD reg,imm | ADD CX,3<br>ADD DI,1AH<br>ADD DL,34H<br>ADD EAX,123456<br>ADD CX,18AFH | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| ADD mem,imm | ADD DATAF,33<br>ADD LIST,'A'<br>ADD DATAM [DI],2<br>ADD BYTE PTR [EAX],3<br>ADD WORD PTR [DI],6A8H | 8086 | 17 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 0000010w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| ADD acc,imm | ADD AX, 3<br>ADD AL,1AH<br>ADD AH,56<br>ADD EAX,3<br>ADD AL,'D' | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## AND Logical AND

001000dw oorrrmmm disp

O D I T S Z A P C
0 * * ? * 0

| Format | Examples | | Clocks |
|---|---|---|---|
| AND reg,reg | AND CX,BX<br>AND DL,BL<br>AND ECX,EBX<br>AND BP,SI<br>AND EDX,EDI | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| AND mem,reg | AND BIT,CH<br>AND LIST,DI<br>AND DATAZ [BX],CL<br>AND [ECX],AL<br>AND [EDX+8*ECX],EDI | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |
| AND reg,mem | AND BL,DATAW<br>AND SI,LIST<br>AND CL,DATAQ [DI]<br>AND CL,[EAX]<br>AND EDX,[EBX+34AH] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

100000sw oo100mmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| AND reg,imm | AND BP,1<br>AND DI,10H<br>AND DL,34H<br>AND EBP,12345<br>AND SP,1234H | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |

| | | | |
|---|---|---|---|
| AND mem,imm | AND DATAD,33<br>AND LIST,4<br>AND DATAS [SI],2<br>AND BYTE PTR [EAX],3<br>AND DWORD PTR [DI],32 | 8086 | 17 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

0010010w data

| Format | Examples | | Clocks |
|---|---|---|---|
| AND acc,imm | AND AX,15<br>AND AL,1FH<br>AND AH,34<br>AND EAX,3<br>AND AL,'R' | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## ARPL Adjust requested privilege level

01100011 oorrrmmm disp

O D I T S Z A P C
*

| Format | Examples | | Clocks |
|---|---|---|---|
| ARPL reg,reg | ARPL AX,BX<br>ARPL BX,SI<br>ARPL CX,DX<br>ARPL BX,AX<br>ARPL DI,SI | 8086 | — |
| | | 8088 | — |
| | | 80286 | 10 |
| | | 80386 | 20 |
| | | 80486 | 9 |
| | | Pentium | 7 |
| ARPL mem,reg | ARPL NUMB,AX<br>ARPL LIST,DI<br>ARPL DATA2 [BX],CX<br>ARPL [ECX],AX<br>ARPL [EDX+4*ECX],DI | 8086 | — |
| | | 8088 | — |
| | | 80286 | 11 |
| | | 80386 | 21 |
| | | 80486 | 9 |
| | | Pentium | 7 |

## BOUND Check array against bounds

01100010 oorrrmmm disp

O D I T S Z A P C

Clocks

| Format | Examples | | |
|---|---|---|---|
| BOUND reg,mem | BOUND AX,BETS<br>BOUND BX,SAID<br>BOUND CX,DATA3<br>BOUND BX,[DI]<br>BOUND DI,[BX+2] | 8086 | — |
| | | 8088 | — |
| | | 80286 | 13 |
| | | 80386 | 10 |
| | | 80486 | 7 |
| | | Pentium | 8 |

## BSF Bit scan forward

00001111 10111100 oorrmmm disp

O D I T S Z A P C

? ? * ? ? ?

Clocks

| Format | Examples | | |
|---|---|---|---|
| BSF reg,reg | BSF AX,BX<br>BSF BX,SI<br>BSF ECX,EBX<br>BSF EBX,EAX<br>BSF DI,SI | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 10 + 3n |
| | | 80486 | 6–42 |
| | | Pentium | 6–43 |
| BSF reg,mem | BSF AX,DATAD<br>BSF BP,LISTG<br>BSF ECX,MEMORY<br>BSF EAX,DATA6<br>BSF DI,[ECX] | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 10 + 3n |
| | | 80486 | 7–43 |
| | | Pentium | 6–43 |

## BSR Bit scan reverse

00001111 10111101 oorrmmm disp

O D I T S Z A P C
? ? * ? ? ?

| Format | Examples | Clocks | |
|---|---|---|---|
| BSR reg,reg | BSR AX,BX<br>BSR BX,SI<br>BSR ECX,EBX<br>BSR EBX,EAX<br>BSR DI,SP | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 10 + 3n |
| | | 80486 | 6–103 |
| | | Pentium | 7–71 |
| BSR reg,mem | BSR AX,DATAE<br>BSR BP,LISTG<br>BSR ECX,MEMORY<br>BSR EAX,DATA6<br>BSR DI,[EBX] | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 10 + 3n |
| | | 80486 | 7–104 |
| | | Pentium | 7–72 |

## BSWAP Byte swap

00001111 11001rrr

O D I T S Z A P C

| Format | Examples | Clocks | |
|---|---|---|---|
| BSWAP reg | BSWAP EAX<br>BSWAP EBX<br>BSWAP ECX<br>BSWAP EDX<br>BSWAP EDI | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | — |
| | | 80486 | 1 |
| | | Pentium | 1 |

## BT Bit test

00001111 10111010 oo100mmm disp data

O D I T S Z A P C *

| Format | Examples | | Clocks |
|---|---|---|---|
| BT reg,imm8 | BT AX,2<br>BT CX,4<br>BT BP,10H<br>BT CX,8<br>BT BX,2 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4 |
| BT mem,imm8 | BT DATA1,2<br>BT LIST,2<br>BT DATA4 [DI],2<br>BT [BX],1<br>BT FROG,3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 4 |

00001111 10100011 disp

| Format | Examples | | Clocks |
|---|---|---|---|
| BT reg,reg | BT AX,CX<br>BT CX,DX<br>BT BP,AX<br>BT SI,CX<br>BT CX,BP | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4 |
| BT mem,reg | BT DATA1,AX<br>BT LIST,DX<br>BT DATA3,CX<br>BT DATA9,BX<br>BT DATA3 [DI],AX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 12 |
| | | 80486 | 8 |
| | | Pentium | 4/9 |

## BTC Bit test and complement

00001111 10111010 oo111mmm disp data

O D I T S Z A P C *

| Format | Examples | | Clocks |
|---|---|---|---|
| BTC reg,imm8 | BTC AX,2<br>BTC CX,4<br>BTC BP,10H<br>BTC CX,8<br>BTC BX,2 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 6 |
| | | Pentium | 7/8 |
| BTC mem,imm8 | DATA1,2<br>BTC LIST,2<br>BTC DATA [DI],3<br>BTC [BX],1<br>BTC TOAD,5 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 8 |
| | | 80486 | 8 |
| | | Pentium | 7/8 |

00001111 10111011 disp

| Format | Examples | | Clocks |
|---|---|---|---|
| BTC reg,reg | BTC AX,CX<br>BTC CX,DX<br>BTC BP,AX<br>BTC SI,CX<br>BTC CX,BX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 6 |
| | | Pentium | 7/13 |
| BTC mem,reg | BTC DATA1,AX<br>BTC LIST,DX<br>BTC DATA3,CX<br>BTC DATA9,BX<br>BTC DATAS [DI],AX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 13 |
| | | 80486 | 13 |
| | | Pentium | 7/13 |

## BTR Bit test and reset

00001111 10111010 oo110mmm disp data

O D I T S Z A P C *

| Format | Examples | | Clocks |
|---|---|---|---|
| BTR reg,imm8 | BTR AX,2<br>BTR CX,4<br>BTR BP,10H<br>BTR CX,8<br>BTR BX,2 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 6 |
| | | Pentium | 7/8 |
| BTR mem,imm8 | BTR DATA1,2<br>BTR LIST,2<br>BTR DATAZ [DI],4<br>BTR [BX],1<br>BTR SLED,6 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 8 |
| | | 80486 | 8 |
| | | Pentium | 7/8 |

00001111 10110011 disp

| Format | Examples | | Clocks |
|---|---|---|---|
| BTR reg,reg | BTR AX,CX<br>BTR CX,DX<br>BTR BP,AX<br>BTR SI,CX<br>BTR BP,CX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 6 |
| | | Pentium | 7/13 |
| BTR mem,reg | BTR DATA1,AX<br>BTR LIST,DX<br>BTR DATA3,CX<br>BTR DATA9,BX<br>BTR DATAD [BX],AX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 13 |
| | | 80486 | 13 |
| | | Pentium | 7/13 |

## BTS Bit test and set

00001111 10111010 oo101mmm disp data — O D I T — S Z A P C*

| Format | Examples | | Clocks |
|---|---|---|---|
| BTS reg,imm8 | BTS AX,2<br>BTS CX,4<br>BTS BP,10H<br>BTS CX,8<br>BTS BX,3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 6 |
| | | Pentium | 7/8 |
| BTS mem,imm8 | BTS DATA1,2<br>BTS LIST,2<br>BTS DATAT [BP],7<br>BTS [BX],1<br>BTS FROG,3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 8 |
| | | 80486 | 8 |
| | | Pentium | 7/8 |

00001111 10101011 disp

| Format | Examples | | Clocks |
|---|---|---|---|
| BTS reg,reg | BTS AX,CX<br>BTS CX,DX<br>BTS BP,AX<br>BTS SI,CX<br>BTS CX,BP | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 6 |
| | | Pentium | 7/13 |
| BTS mem,reg | BTS DATA1,AX<br>BTS LIST,DX<br>BTS DATA3,CX<br>BTS DATA9,BX<br>BTS DATAL [BP],AX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 13 |
| | | 80486 | 13 |
| | | Pentium | 7/13 |

## CALL Call procedure (subroutine)

| 11101000 disp Format | Examples | O D I T S Z A P C Clocks | |
|---|---|---|---|
| CALL label (near) | CALL FORFUN<br>CALL HOME<br>CALL ET<br>CALL WAITING<br>CALL SOMEONE | 8086 | 19 |
| | | 8088 | 23 |
| | | 80286 | 7 |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 1 |
| 10011010 disp Format | Examples | | Clocks |
| CALL label (far) | CALL FAR PTR DATES<br>CALL WHAT<br>CALL WHERE<br>CALL FARCE<br>CALL WHOM | 8086 | 28 |
| | | 8088 | 36 |
| | | 80286 | 13 |
| | | 80386 | 17 |
| | | 80486 | 18 |
| | | Pentium | 4 |
| 11111111 oo010mmm Format | Examples | | Clocks |
| CALL reg (near) | CALL AX<br>CALL BX<br>CALL CX<br>CALL DI<br>CALL SI | 8086 | 16 |
| | | 8088 | 20 |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 5 |
| | | Pentium | 2 |
| CALL mem (near) | CALL ADDRESS<br>CALL NEAR PTR [DI]<br>CALL DATA<br>CALL FROG<br>CALL HERO | 8086 | 21 + ea |
| | | 8088 | 29 + ea |
| | | 80286 | 11 |
| | | 80386 | 10 |
| | | 80486 | 5 |
| | | Pentium | 2 |

| 11111111 oo011mmm<br>Format | Examples | | Clocks |
|---|---|---|---|
| CALL mem<br>(far) | CALL FAR LIST [SI]<br>CALL FROM HERE<br>CALL TO THERE<br>CALL SIXX<br>CALL OCT | 8086 | 37 + ea |
| | | 8088 | 53 + ea |
| | | 80286 | 16 |
| | | 80386 | 22 |
| | | 80486 | 17 |
| | | Pentium | 5 |

**CBW** Convert byte to word

| 10011000<br>Example | O D I T S Z A P C<br>Clocks |
|---|---|
| CBW | 8086 — 2 |
| | 8088 — 2 |
| | 80286 — 2 |
| | 80386 — 3 |
| | 80486 — 3 |
| | Pentium — 3 |

**CDQ** Convert doubleword to quadword

| 10011001<br>Example | O D I T S Z A P C<br>Clocks |
|---|---|
| CDQ | 8086 — — |
| | 8088 — — |
| | 80286 — — |
| | 80386 — 2 |
| | 80486 — 2 |
| | Pentium — 2 |

## CLC Clear carry flag

11111000

O D I T S Z A P C
0

Example

Clocks

| Example | | |
|---|---|---|
| CLC | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 2 |
| | Pentium | 2 |

## CLD Clear direction flag

11111100

O D I T S Z A P C
0

Example

Clocks

| Example | | |
|---|---|---|
| CLD | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 2 |
| | Pentium | 2 |

## CLI Clear interrupt flag

11111010

O D I T S Z A P C
0

Example

Clocks

| Example | | |
|---|---|---|
| CLI | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 3 |
| | 80386 | 3 |
| | 80486 | 5 |
| | Pentium | 7 |

## CLTS Clear task switched flag (CR0)

00001111 00000110 — O D I T — S Z A P C

Example

| | | Clocks |
|---|---|---|
| CLTS | 8086 | — |
| | 8088 | — |
| | 80286 | 2 |
| | 80386 | 5 |
| | 80486 | 7 |
| | Pentium | 10 |

## CMC Complement carry flag

10011000 — O D I T — S Z A P C
*

Example

| | | Clocks |
|---|---|---|
| CMC | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 2 |
| | Pentium | 2 |

## CMP Compare operands

001110dw oorrrmmm disp — O D I T — S Z A P C
* * * * * *

| Format | Examples | | Clocks |
|---|---|---|---|
| CMP reg,reg | CMP AX,BX<br>CMP AL,BL<br>CMP EAX,EBX<br>CMP CX,SI<br>CMP ESI,EDI | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |

| | | | |
|---|---|---|---|
| CMP mem,reg | CMP DATAF,AL<br>CMP LIST,SI<br>CMP DATAE [BX],CL<br>CMP [EAX],AL<br>CMP [EBX+2*ECX],EBX | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 5 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |
| CMP reg,mem | CMP BL,DATAZ<br>CMP SI,LIST<br>CMP CL,DATAD [DI]<br>CMP CL,[EAX]<br>CMP EDX,[EBX+200H] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 6 |
| | | 80386 | 6 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |
| 100000sw oo111mmm disp data<br>Format | Examples | | Clocks |
| CMP reg,imm | CMP CX,3<br>CMP DI,1AH<br>CMP DL,34H<br>CMP EBX,12345<br>CMP CX,123AH | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| CMP mem,imm | CMP DATAS,33<br>CMP LIST,'A'<br>CMP DATAF [DI],87H<br>CMP BYTE PTR [EAX],3<br>CMP WORD PTR [SI],7 | 8086 | 10 + ea |
| | | 8088 | 14 + ea |
| | | 80286 | 6 |
| | | 80386 | 5 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

| 0011110w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| CMP acc,imm | CMP AX,3<br>CMP AL,1AH<br>CMP AH,34<br>CMP EAX,3<br>CMP AL,'I' | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## CMPS Compare strings

| 1010011w | | O D I T | S Z A P C | |
|---|---|---|---|---|
| | | * | * * * * * | |
| Format | Examples | | | Clocks |
| CMPSB<br>CMPSW<br>CMPSD | CMPSB<br>CMPSW<br>CMPSD<br>CMPS DATA1<br>REPE CMPSB<br>REPNE CMPSW | 8086 | | 22 |
| | | 8088 | | 30 |
| | | 80286 | | 8 |
| | | 80386 | | 10 |
| | | 80486 | | 8 |
| | | Pentium | | 5 |

## CMPXCHG Compare and exchange

| 00001111 1011000w 11rrrrrr | | O D I T | S Z A P C | |
|---|---|---|---|---|
| | | * | * * * * * | |
| Format | Examples | | | Clocks |
| CMPXCHG<br>reg,reg | CMPXCHG EAX,EBX<br>CMPXCHG ECX,EDX | 8086 | | — |
| | | 8088 | | — |
| | | 80286 | | — |
| | | 80386 | | — |
| | | 80486 | | 6 |
| | | Pentium | | 6 |

00001111 1011000w oorrrmmm

| Format | Examples | | Clocks |
|---|---|---|---|
| CMPXCHG mem,reg | CMPXCHG DATAD,EAX<br>CMPXCHG DATA2,EBX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | — |
| | | 80486 | 7 |
| | | Pentium | 6 |

**CMPXCHG8B** Compare and exchange 8 bytes

00001111 11000111 11rrrrrr

O D I T S Z A P C
*

| Format | Examples | | Clocks |
|---|---|---|---|
| CMPXCHG mem64 | CMPXCHG8B DATA3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | — |
| | | 80486 | — |
| | | Pentium | 10 |

**CPUID** CPU identification code to EAX

00001111 10100010

O D I T S Z A P C

| Example | | Clocks |
|---|---|---|
| CPUID | 8086 | — |
| | 8088 | — |
| | 80286 | — |
| | 80386 | — |
| | 80486 | — |
| | Pentium | 14 |

## CWD Convert word to doubleword

10011001
Example

O D I T S Z A P C
Clocks

| | |
|---|---|
| 8086 | 5 |
| 8088 | 5 |
| 80286 | 2 |
| 80386 | 2 |
| 80486 | 3 |
| Pentium | 2 |

## CWDE Convert word to extended double word

10011000
Example

O D I T S Z A P C
Clocks

| | |
|---|---|
| 8086 | — |
| 8088 | — |
| 80286 | — |
| 80386 | 3 |
| 80486 | 3 |
| Pentium | 3 |

## DAA Decimal adjust AL after addition

00100111
Example

O D I T S Z A P C
? * * * * *
Clocks

| | |
|---|---|
| 8086 | 4 |
| 8088 | 4 |
| 80286 | 3 |
| 80386 | 4 |
| 80486 | 2 |
| Pentium | 3 |

## DAS Decimal adjust AL after subtraction

00101111

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| ? | | | | * | * | * | * | * |

Example

| Example | 8086 | 4 |
|---|---|---|
| | 8088 | 4 |
| | 80286 | 3 |
| | 80386 | 4 |
| | 80486 | 2 |
| | Pentium | 3 |

## DEC Decrement

1111111w oo001mmm disp

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| * | | | | * | * | * | * | |

| Format | Examples | | Clocks |
|---|---|---|---|
| DEC reg8 | DEC BL<br>DEC BH<br>DEC CL<br>DEC DH<br>DEC AH | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| DEC mem | DEC DATAR<br>DEC LIST<br>DEC DATA [SI]<br>DEC BYTE PTR [EAX]<br>DEC WORD PTR [DI] | 8086 | 15 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 01001rrr<br>Format | Examples | | Clocks |
|---|---|---|---|
| DEC reg16<br>DEC reg32 | DEC AX<br>DEC EAX<br>DEC CX<br>DEC EBX<br>DEC DI | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## **DIV** Unsigned division

| 1111011w oo110mmm disp<br>Format | Examples | O D I T<br>? | S Z A P C<br>? ? ? ? ?<br>Clocks |
|---|---|---|---|
| DIV reg | DIV BL<br>DIV BH<br>DIV ECX<br>DIV BH<br>DIV CH | 8086 | 162 |
| | | 8088 | 162 |
| | | 80286 | 22 |
| | | 80386 | 38 |
| | | 80486 | 40 |
| | | Pentium | 17–41 |
| DIV mem | DIV DATA2<br>DIV LIST<br>DIV DATA3 [DI]<br>DIV BYTE PTR [EAX]<br>DIV WORD PTR [DI] | 8086 | 168 |
| | | 8088 | 176 |
| | | 80286 | 25 |
| | | 80386 | 41 |
| | | 80486 | 40 |
| | | Pentium | 17–41 |

## ENTER Create a stack frame

| 11001000 data Format | Examples | O D I T S Z A P C | Clocks |
|---|---|---|---|
| ENTER imm,0 | ENTER 4,0<br>ENTER 8,0<br>ENTER 100,0<br>ENTER 200,0<br>ENTER 1024,0 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 11 |
| | | 80386 | 10 |
| | | 80486 | 14 |
| | | Pentium | 11 |
| ENTER imm,1 | ENTER 4,1<br>ENTER 10,1 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 15 |
| | | 80386 | 12 |
| | | 80486 | 17 |
| | | Pentium | 15 |
| ENTER imm,imm | ENTER 3,6<br>ENTER 100,3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 12 |
| | | 80386 | 15 |
| | | 80486 | 17 |
| | | Pentium | 15 |

## ESC Escape

11011nnn oonnnmmm
nnnnnn = opcode for coprocessor

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| ESC imm,reg | ESC 5,AL<br>ESC 5,BH<br>ESC 6,CH<br>FADD ST,ST(3) | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 20 |
| | | 80386 | var |
| | | 80486 | var |
| | | Pentium | var |
| ESC imm,mem | ESC 2,DATAS<br>ESC 3,FROG<br>FADD DATAE<br>FMUL FROG | 8086 | 8 + ea |
| | | 8088 | 12 + ea |
| | | 80286 | 20 |
| | | 80386 | var |
| | | 80486 | var |
| | | Pentium | var |

## HLT Halt

11110100

O D I T S Z A P C

| Example | | Clocks |
|---|---|---|
| HLT | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 5 |
| | 80486 | 4 |
| | Pentium | ∞ |

## IDIV Signed division

1111011w oo111mmm disp

O D I T S Z A P C
? ? ? ? ? ? ? ? ?

| Format | Examples | | Clocks |
|---|---|---|---|
| IDIV reg | IDIV BL<br>IDIV BH<br>IDIV ECX<br>IDIV BH<br>IDIV CX | 8086 | 184 |
| | | 8088 | 184 |
| | | 80286 | 25 |
| | | 80386 | 43 |
| | | 80486 | 43 |
| | | Pentium | 22–46 |
| IDIV mem | IDIV DATAF<br>IDIV LIST<br>IDIV DATAG [DI]<br>IDIV BYTE PTR [EAX]<br>IDIV WORD PTR [DI] | 8086 | 190 |
| | | 8088 | 194 |
| | | 80286 | 28 |
| | | 80386 | 46 |
| | | 80486 | 44 |
| | | Pentium | 22–46 |

## IMUL Signed multiplication

1111011w oo101mmm disp

O D I T S Z A P C
* ? ? ? ? *

| Format | Examples | | Clocks |
|---|---|---|---|
| IMUL reg | IMUL BL<br>IMUL CL<br>IMUL CX<br>IMUL ECX<br>IMUL EBX | 8086 | 154 |
| | | 8088 | 154 |
| | | 80286 | 21 |
| | | 80386 | 38 |
| | | 80486 | 42 |
| | | Pentium | 10–11 |

| IMUL mem | IMUL DATAN<br>IMUL LIST<br>IMUL DATAK [SI]<br>IMUL BYTE PTR [EAX]<br>IMUL WORD PTR [DI] | 8086 | 160 |
|---|---|---|---|
| | | 8088 | 164 |
| | | 80286 | 24 |
| | | 80386 | 41 |
| | | 80486 | 42 |
| | | Pentium | 10–11 |

| 011010sl oorrrmmm disp data<br>Format | Examples | | Clocks |
|---|---|---|---|
| IMUL reg,imm | IMUL CX,16<br>IMUL DX,100<br>IMUL EAX,20 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 21 |
| | | 80386 | 38 |
| | | 80486 | 42 |
| | | Pentium | 10 |
| IMUL reg,reg,imm | IMUL DX,AX,2<br>IMUL CX,DX,3<br>IMUL BX,AX,33 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 21 |
| | | 80386 | 38 |
| | | 80486 | 42 |
| | | Pentium | 10 |
| IMUL reg,mem,imm | IMUL CX,DATA,4 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 24 |
| | | 80386 | 38 |
| | | 80486 | 42 |
| | | Pentium | 10 |

| 00001111 10101111 oorrrmmm disp<br>Format | Examples | | Clocks |
|---|---|---|---|
| IMUL reg,reg | IMUL CX,DX<br>IMUL DX,BX<br>IMUL EAX,ECX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 38 |
| | | 80486 | 42 |
| | | Pentium | 10 |
| IMUL reg,mem | IMUL DX,DATA<br>IMUL CX,FROG<br>IMUL BX,LISTS | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 41 |
| | | 80486 | 42 |
| | | Pentium | 10 |

**IN** Input data from port

| 1110010w port number<br>Format | Examples | O D I T S Z A P C<br>Clocks | |
|---|---|---|---|
| IN acc,pt | IN AL,12H<br>IN AX,12H<br>IN AL,0FFH<br>IN AX,0FFH<br>IN EAX,10H | 8086 | 10 |
| | | 8088 | 14 |
| | | 80286 | 5 |
| | | 80386 | 12 |
| | | 80486 | 14 |
| | | Pentium | 7 |

| 1110110w<br>Format | Examples | | Clocks |
|---|---|---|---|
| IN acc,DX | IN AL,DX<br>IN AX,DX<br>IN EAX,DX | 8086 | 8 |
| | | 8088 | 12 |
| | | 80286 | 5 |
| | | 80386 | 13 |
| | | 80486 | 14 |
| | | Pentium | 7 |

## INC Increment

| 1111111w oo000mmm disp | | O D I T<br>* | S Z A P C<br>* * * * * |
|---|---|---|---|
| Format | Examples | | Clocks |
| INC reg8 | INC BL<br>INC BH<br>INC CL<br>INC DH<br>INC AH | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| INC mem | INC DATA2<br>INC LIST<br>INC DATA3 [BX]<br>INC BYTE PTR [EAX]<br>INC WORD PTR [BX]<br>INC DWORD PTR [ECX] | 8086 | 15 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 01000rrr<br>Format | Examples | | Clocks |
|---|---|---|---|
| INC reg16<br>INC reg32 | INC AX<br>INC EAX<br>INC CX<br>INC EBX<br>INC DI | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

**INS** Input string from port

| 0110110w<br>Format | Examples | | O D I T S Z A P C<br>Clocks |
|---|---|---|---|
| INSB<br>INSW<br>INSD | INSB<br>INSW<br>INSD<br>INS DATA<br>REP INSB | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 |
| | | 80386 | 15 |
| | | 80486 | 17 |
| | | Pentium | 9 |

**INT** Interrupt

| 11001101 type<br>Format | Examples | | O D I T S Z A P C<br>Clocks |
|---|---|---|---|
| INT type | INT 10H<br>INT 255<br>INT 21H<br>INT 20H<br>INT 15H | 8086 | 51 |
| | | 8088 | 71 |
| | | 80286 | 23 |
| | | 80386 | 37 |
| | | 80486 | 30 |
| | | Pentium | 16 |

| 11001100<br>Example | | Clocks |
|---|---|---|
| INT 3 | 8086 | 52 |
| | 8088 | 72 |
| | 80286 | 23 |
| | 80386 | 33 |
| | 80486 | 26 |
| | Pentium | 13 |

**INTO** Interrupt on overflow

| 11001110<br>Example | O D I T S Z A P C<br>Clocks | |
|---|---|---|
| INTO | 8086 | 53 |
| | 8088 | 73 |
| | 80286 | 24 |
| | 80386 | 35 |
| | 80486 | 28 |
| | Pentium | 27 |

**INVD** Invalidate data cache

| 00001111 00001000<br>Example | O D I T S Z A P C<br>Clocks | |
|---|---|---|
| INVD | 8086 | — |
| | 8088 | — |
| | 80286 | — |
| | 80386 | — |
| | 80486 | 4 |
| | Pentium | 15 |

## INVLPG Invalidate TLB entry

00001111 00000001 oo111mmm

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| INVLPG mem | INVLPG DATA<br>INVLPG LIST | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | — |
| | | 80486 | 12 |
| | | Pentium | 25 |

## IRET/IRETD Interrupt return

11001101 data

O D I T S Z A P C
* * * * * * * * *

| Format | Examples | | Clocks |
|---|---|---|---|
| IRET<br>IRETD | IRET<br>IRETD<br>IRET 10H | 8086 | 32 |
| | | 8088 | 44 |
| | | 80286 | 17 |
| | | 80386 | 22 |
| | | 80486 | 15 |
| | | Pentium | 8 |

## Jconditional Conditional jump

0111cccc disp

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| Jcc label<br>(8-bit disp) | JA BELOW<br>JB ABOVE<br>JG GREATER<br>JE EQUAL<br>JZ ZERO | 8086 | 16/4 |
| | | 8088 | 16/4 |
| | | 80286 | 7/3 |
| | | 80386 | 7/3 |
| | | 80486 | 3/1 |
| | | Pentium | 1 |

00001111 1000cccc disp

| Format | Examples | | Clocks |
|---|---|---|---|
| Jcc label<br>(16-bit disp) | JNE NOT_MORE<br>JLE LESS_THAN | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7/3 |
| | | 80486 | 3/1 |
| | | Pentium | 1 |

| Condition Codes | Mnemonic | Flag | Description |
|---|---|---|---|
| 0000 | JO | O = 1 | Jump if overflow |
| 0001 | JNO | O = 0 | Jump if no overflow |
| 0010 | JB/JNAE | C = 1 | Jump if below |
| 0011 | JAE/JNB | C = 0 | Jump if above or equal |
| 0100 | JE/JZ | Z = 1 | Jump if equal/zero |
| 0101 | JNE/JNZ | Z = 0 | Jump if not equal/not zero |
| 0110 | JBE/JNA | C = 1 + Z = 1 | Jump if below or equal |
| 0111 | JA/JNBE | C = 0 • Z = 0 | Jump if above |
| 1000 | JS | S = 1 | Jump if sign |
| 1001 | JNS | S = 0 | Jump if no sign |
| 1010 | JP/JPE | P = 1 | Jump if parity even |
| 1011 | JNP/JPO | P = 0 | Jump if parity odd |
| 1100 | JL/JNGE | S • O | Jump if less than |
| 1101 | JGE/JNL | S = O | Jump if greater or equal |
| 1110 | JLE/JNG | Z = 1 + S • O | Jump if less than or equal |
| 1111 | JG/JNLE | Z = 0 + S = O | Jump if greater |

## **JCXZ/JECXZ** Jump if CX (ECX) equals zero

11100011 O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| JCXZ label<br>JECXZ label | JCXZ LOTSA<br>JCXZ OVER<br>JECXZ UPPER<br>JECXZ UNDER<br>JCXZ NEXT | 8086 | 18/6 |
| | | 8088 | 18/6 |
| | | 80286 | 8/4 |
| | | 80386 | 9/5 |
| | | 80486 | 8/5 |
| | | Pentium | 6/5 |

**JMP** Unconditional jump

| 11101011 disp Format | Examples | O D I T S Z A P C Clocks | |
|---|---|---|---|
| JMP label (short) | JMP SHORT UP<br>JMP SHORT DOWN<br>JMP SHORT OVER<br>JMP SHORT CIRCUIT<br>JMP SHORT ARM | 8086 | 15 |
| | | 8088 | 15 |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1 |

| 11101001 disp Format | Examples | Clocks | |
|---|---|---|---|
| JMP label (near) | JMP VER<br>JMP FROG<br>JMP UNDER<br>JMP NEAR PTR OVER | 8086 | 15 |
| | | 8088 | 15 |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1 |

| 11101010 disp Format | Examples | Clocks | |
|---|---|---|---|
| JMP label (far) | JMP VER<br>JMP FROG<br>JMP UNDER<br>JMP FAR PTR THERE | 8086 | 15 |
| | | 8088 | 15 |
| | | 80286 | 11 |
| | | 80386 | 12 |
| | | 80486 | 17 |
| | | Pentium | 3 |

| 11111111 oo100mmm<br>Format | Examples | | Clocks |
|---|---|---|---|
| JMP reg<br>(near) | JMP AX<br>JMP EAX<br>JMP CX<br>JMP DX | 8086 | 11 |
| | | 8088 | 11 |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 2 |
| JMP mem<br>(near) | JMP DATA<br>JMP LIST<br>JMP DATA [DI+2] | 8086 | 18 + ea |
| | | 8088 | 18 + ea |
| | | 80286 | 11 |
| | | 80386 | 10 |
| | | 80486 | 5 |
| | | Pentium | 4 |

| 11111111 oo101mmm<br>Format | Examples | | Clocks |
|---|---|---|---|
| JMP mem<br>(far) | JMP WAY_OFF<br>JMP TABLE<br>JMP UP | 8086 | 24 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 15 |
| | | 80386 | 12 |
| | | 80486 | 13 |
| | | Pentium | 4 |

**LAHF** Load AH from flags

| 10011111<br>Example | O D I T S Z A P C<br>Clocks | |
|---|---|---|
| LAHF | 8086 | 4 |
| | 8088 | 4 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 3 |
| | Pentium | 2 |

## LAR — Load access rights byte

00001111 00000010 oorrrmmm disp

O D I T S Z A P C
*

| Format | Examples | | Clocks |
|---|---|---|---|
| LAR reg,reg | LAR AX,BX<br>LAR CX,DX<br>LAR EAX,ECX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 14 |
| | | 80386 | 15 |
| | | 80486 | 11 |
| | | Pentium | 8 |
| LAR reg,mem | LAR CX,DATA<br>LAR AX,LIST<br>LAR ECX,FROG | 8086 | — |
| | | 8088 | — |
| | | 80286 | 16 |
| | | 80386 | 16 |
| | | 80486 | 11 |
| | | Pentium | 8 |

## LDS — Load far pointer

11000101 oorrrmmm

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| LDS reg,mem | LDS DI,DATA<br>LDS SI,LIST<br>LDS BX,ARRAY<br>LDS CX,PNTR | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 6 |
| | | Pentium | 4 |

## LES Load far pointer

11000100 oorrrmmm

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| LES reg,mem | LES DI,DATA<br>LES SI,LIST<br>LES BX,ARRAY<br>LES CX,PNTR | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 6 |
| | | Pentium | 4 |

## LFS Load far pointer

00001111 10110100 oorrrmmm disp

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| LFS reg,mem | LFS DI,DATA<br>LFS SI,LIST<br>LFS BX,ARRAY<br>LFS CX,PNTR | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 6 |
| | | Pentium | 4 |

## LGS Load far pointer

00001111 10110101 oorrrmmm disp

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| LGS reg,mem | LGS DI,DATA<br>LGS SI,LIST<br>LGS BX,ARRAY<br>LGS CX,PNTR | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 6 |
| | | Pentium | 4 |

## LSS Load far pointer

00001111 10110010 oorrrmmm disp
Format: LSS reg,mem
Examples: LSS DI,DATA; LSS SI,LIST; LSS BX,ARRAY; LSS CX,PNTR

O D I T S Z A P C

| Clocks |
|---|
| 8086 — |
| 8088 — |
| 80286 — |
| 80386 7 |
| 80486 6 |
| Pentium 4 |

## LEA Load effective address

10001101 oorrrmmm disp
Format: LEA reg,mem
Examples: LEA DI,DATA1; LEA SI,LIST; LEA BX,ARRAY; LEA CX,PNTR; LEA BP,ADDR

O D I T S Z A P C

| Clocks |
|---|
| 8086 2 + ea |
| 8088 2 + ea |
| 80286 3 |
| 80386 2 |
| 80486 2 |
| Pentium 1 |

## LEAVE Leave high-level procedure

11001001
Example: LEAVE

O D I T S Z A P C

| Clocks |
|---|
| 8086 — |
| 8088 — |
| 80286 5 |
| 80386 4 |
| 80486 5 |
| Pentium 3 |

## LGDT Load global descriptor table

00001111 00000001 oo010mmm disp
Format | Examples | O D I T | S Z A P C Clocks

| Format | Examples | | |
|---|---|---|---|
| LGDT mem64 | LGDT DESCRIP<br>LGDT TABLE | 8086 | — |
| | | 8088 | — |
| | | 80286 | 11 |
| | | 80386 | 11 |
| | | 80486 | 11 |
| | | Pentium | 6 |

## LIDT Load interrupt descriptor table

00001111 00000001 oo011mmm disp
Format | Examples | O D I T | S Z A P C Clocks

| Format | Examples | | |
|---|---|---|---|
| LIDT mem64 | LIDT DATA<br>LIDT DESCRIP | 8086 | — |
| | | 8088 | — |
| | | 80286 | 12 |
| | | 80386 | 11 |
| | | 80486 | 11 |
| | | Pentium | 6 |

## LLDT Load local descriptor table

00001111 00000000 oo010mmm disp
Format | Examples | O D I T | S Z A P C Clocks

| Format | Examples | | |
|---|---|---|---|
| LLDT reg | LLDT AX<br>LLDT CX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 17 |
| | | 80386 | 20 |
| | | 80486 | 11 |
| | | Pentium | 9 |

| LLDT mem | LLDT DATA2<br>LLDT LIST | 8086 | — |
|---|---|---|---|
| | | 8088 | — |
| | | 80286 | 19 |
| | | 80386 | 24 |
| | | 80486 | 11 |
| | | Pentium | 9 |

**LMSW** Load machine status word

00001111 00000001 oo110mmm disp
should only be used with the 80286

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| LMSW reg | LMSW AX<br>LMSW CX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 3 |
| | | 80386 | 10 |
| | | 80486 | 2 |
| | | Pentium | 8 |
| LMSW mem | LMSW DATA4<br>LMSW LIST | 8086 | — |
| | | 8088 | — |
| | | 80286 | 6 |
| | | 80386 | 13 |
| | | 80486 | 3 |
| | | Pentium | 8 |

**LOCK** Lock the bus

11110000

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| LOCK inst | LOCK:XCHG AX,BX<br>LOCK:MOV AL,AH | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 0 |
| | | 80386 | 0 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## LODS Load string operand

| 1010110w Format | Examples | O D I T S Z A P C | Clocks |
|---|---|---|---|
| LODSB<br>LODSW<br>LODSD | LODSB<br>LODSW<br>LODSD<br>LODS DATA5<br>LODS ES:DATA6 | 8086 | 12 |
| | | 8088 | 16 |
| | | 80286 | 5 |
| | | 80386 | 5 |
| | | 80486 | 5 |
| | | Pentium | 2 |

## LOOP Loop until CX = 0

| 11100010 disp Format | Examples | O D I T S Z A P C | Clocks |
|---|---|---|---|
| LOOP label | LOOP DATA_FROG<br>LOOP BACK | 8086 | 17/5 |
| | | 8088 | 17/5 |
| | | 80286 | 8/4 |
| | | 80386 | 11 |
| | | 80486 | 7/6 |
| | | Pentium | 5/6 |

## LOOPE Loop while equal

| 11100001 disp Format | Examples | O D I T S Z A P C | Clocks |
|---|---|---|---|
| LOOPE label<br>LOOPZ label | LOOPE NEXT<br>LOOPE AGAIN<br>LOOPZ REPEAT | 8086 | 18/6 |
| | | 8088 | 18/6 |
| | | 80286 | 8/4 |
| | | 80386 | 11 |
| | | 80486 | 9/6 |
| | | Pentium | 7/8 |

## LOOPNE Loop while not equal

11100000 disp

| Format | Examples | O D I T S Z A P C | Clocks |
|---|---|---|---|
| LOOPNE label<br>LOOPNZ label | LOOPNE AGAIN<br>LOOPNE BACK<br>LOOPNZ REPL | 8086 | 19/5 |
| | | 8088 | 19/5 |
| | | 80286 | 8/4 |
| | | 80386 | 11 |
| | | 80486 | 9/6 |
| | | Pentium | 7/8 |

## LSL Load segment limit

00001111 00000011 oorrrmmm disp

O D I T S Z A P C
*

| Format | Examples | | Clocks |
|---|---|---|---|
| LSL reg,reg | LSL AX,BX<br>LSL CX,BX<br>LSL DX,AX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 14 |
| | | 80386 | 25 |
| | | 80486 | 10 |
| | | Pentium | 8 |
| LSL reg,mem | LSL AX,LIMIT<br>LSL EAX,NUMB | 8086 | — |
| | | 8088 | — |
| | | 80286 | 16 |
| | | 80386 | 26 |
| | | 80486 | 10 |
| | | Pentium | 8 |

## LTR Load task register

00001111 00000000 oo001mmm disp

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| LTR reg | LTR AX<br>LTR CX<br>LTR DX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 17 |
| | | 80386 | 23 |
| | | 80486 | 20 |
| | | Pentium | 10 |
| LTR mem | LTR TASK<br>LTR EDGE | 8086 | — |
| | | 8088 | — |
| | | 80286 | 19 |
| | | 80386 | 27 |
| | | 80486 | 20 |
| | | Pentium | 10 |

## MOV Move data

100010dw oorrrmmm disp

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| MOV reg,reg | MOV CL,CH<br>MOV BH,CL<br>MOV CX,DX<br>MOV EAX,ECX<br>MOV EBP,ESI | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |
| MOV mem,reg | MOV DATA7,DL<br>MOV NUMB,CX<br>MOV TEMP,EBX<br>MOV TEMP1,CH<br>MOV DATA2,CL | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

| | | | |
|---|---|---|---|
| MOV reg,mem | MOV DL,DATA42<br>MOV DX,NUMB<br>MOV EBX,TEMP<br>MOV CH,TEMP1<br>MOV CL,DATA2 | 8086 | 10 + ea |
| | | 8088 | 12 + ea |
| | | 80286 | 5 |
| | | 80386 | 4 |
| | | 80486 | 1 |
| | | Pentium | 1 |

| 1100011w oo000mmm disp data<br>Format | Examples | | Clocks |
|---|---|---|---|
| MOV mem,imm | MOV DATAF,23H<br>MOV LIST,12H<br>MOV BYTE PTR [DI],2<br>MOV NUMB,234H<br>MOV DWORD PTR[SI],100 | 8086 | 10 + ea |
| | | 8088 | 14 + ea |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

| 1011wrrr data<br>Format | Examples | | Clocks |
|---|---|---|---|
| MOV reg,imm | MOV BX,23H<br>MOV CX,12H<br>MOV CL,2<br>MOV ECX,123423H<br>MOV DI,100 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

| 101000dw disp<br>Format | Examples | | Clocks |
|---|---|---|---|
| MOV mem,acc | MOV DATA_FOO,AL<br>MOV NUMB,AX<br>MOV NUMB1,EAX | 8086 | 10 |
| | | 8088 | 14 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

| | | | |
|---|---|---|---|
| MOV acc,mem | MOV AL,DATAE<br>MOV AX,NUMB<br>MOV EAX,TEMP | 8086 | 10 |
| | | 8088 | 14 |
| | | 80286 | 5 |
| | | 80386 | 4 |
| | | 80486 | 1 |
| | | Pentium | 1 |

100011d0 oosssmmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| MOV seg,reg | MOV SS,AX<br>MOV DS,DX<br>MOV ES,CX | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |
| MOV seg,mem | MOV SS,DATA9<br>MOV DS,NUMB<br>MOV ES,TEMP1 | 8086 | 8 + ea |
| | | 8088 | 12 + ea |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 2/3 |
| MOV reg,seg | MOV AX,DS<br>MOV DX,ES<br>MOV CX,CS | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

| | | | |
|---|---|---|---|
| MOV mem,seg | MOV DATAA,SS<br>MOV NUMB,ES<br>MOV TEMP1,DS | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |
| 00001111 001000d0 11rrrmmm<br>Format | Examples | | Clocks |
| MOV reg,cr | MOV EAX,CR0<br>MOV EBX,CR2<br>MOV ECX,CR3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 6 |
| | | 80486 | 4 |
| | | Pentium | 4 |
| MOV cr,reg | MOV CR0,EAX<br>MOV CR2,EBX<br>MOV CR3,ECX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 10 |
| | | 80486 | 4 |
| | | Pentium | 12–46 |
| 00001111 001000d1 11rrrmmm<br>Format | Examples | | Clocks |
| MOV reg,dr | MOV EBX,DR6<br>MOV EAX,DR6<br>MOV EDX,DR1 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 22 |
| | | 80486 | 10 |
| | | Pentium | 11 |

| | | | |
|---|---|---|---|
| MOV dr,reg | MOV DR1,ECX<br>MOV DR2,ESI<br>MOV DR6,EBP | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 22 |
| | | 80486 | 11 |
| | | Pentium | 11 |

00001111 001001d0 11rrrmmm

| Format | Examples | | Clocks |
|---|---|---|---|
| MOV reg,tr | MOV EAX,TR6<br>MOV EDX,TR7 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 12 |
| | | 80486 | 4 |
| | | Pentium | 11 |
| MOV tr,seg | MOV TR6,EDX<br>MOV TR7,ESI | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 12 |
| | | 80486 | 6 |
| | | Pentium | 11 |

## MOVS Move string data

1010010w

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| MOVSB<br>MOVSW<br>MOVSD | MOVSB<br>MOVSW<br>MOVSD<br>MOVS DAT1,DAT2<br>REP MOVSB | 8086 | 18 |
| | | 8088 | 26 |
| | | 80286 | 5 |
| | | 80386 | 7 |
| | | 80486 | 7 |
| | | Pentium | 4 |

## MOVSX Move with sign extend

00001111 1011111w oorrrmmm disp

| Format | Examples | O | D | I | T | S | Z | A | P | C Clocks |
|---|---|---|---|---|---|---|---|---|---|---|
| MOVSX reg,reg | MOVSX BX,AL<br>MOVSX EAX,DX | | | | | | | | | |
| | | 8086 | | | | | | | | — |
| | | 8088 | | | | | | | | — |
| | | 80286 | | | | | | | | — |
| | | 80386 | | | | | | | | 3 |
| | | 80486 | | | | | | | | 3 |
| | | Pentium | | | | | | | | 3 |
| MOVSX reg,mem | MOVSX AX,DATA34<br>MOVSX EAX,NUMB | | | | | | | | | |
| | | 8086 | | | | | | | | — |
| | | 8088 | | | | | | | | — |
| | | 80286 | | | | | | | | — |
| | | 80386 | | | | | | | | 6 |
| | | 80486 | | | | | | | | 3 |
| | | Pentium | | | | | | | | 3 |

## MOVZX Move with zero extend

00001111 1011011w oorrrmmm disp

| Format | Examples | O | D | I | T | S | Z | A | P | C Clocks |
|---|---|---|---|---|---|---|---|---|---|---|
| MOVZX reg,reg | MOVZX BX,AL<br>MOVZX EAX,DX | | | | | | | | | |
| | | 8086 | | | | | | | | — |
| | | 8088 | | | | | | | | — |
| | | 80286 | | | | | | | | — |
| | | 80386 | | | | | | | | 3 |
| | | 80486 | | | | | | | | 3 |
| | | Pentium | | | | | | | | 3 |
| MOVZX reg,mem | MOVZX AX,DATADE<br>MOVZX EAX,NUMB | | | | | | | | | |
| | | 8086 | | | | | | | | — |
| | | 8088 | | | | | | | | — |
| | | 80286 | | | | | | | | — |
| | | 80386 | | | | | | | | 6 |
| | | 80486 | | | | | | | | 3 |
| | | Pentium | | | | | | | | 3 |

## MUL Unsigned multiplication

1111011w oo100mmm disp

O D I T S Z A P C
* ? ? ? ? *

| Format | Examples | | Clocks |
|---|---|---|---|
| MUL reg | MUL BL<br>MUL CX<br>MUL ECX | 8086 | 118 |
| | | 8088 | 143 |
| | | 80286 | 21 |
| | | 80386 | 38 |
| | | 80486 | 42 |
| | | Pentium | 11 |
| MUL mem | MUL DATA69<br>MUL BYTE PTR [SI]<br>MUL WORD PTR [SI]<br>MUL DWORD PTR [ECX] | 8086 | 139 |
| | | 8088 | 143 |
| | | 80286 | 24 |
| | | 80386 | 41 |
| | | 80486 | 42 |
| | | Pentium | 11 |

## NEG Negate

1111011w oo011mmm disp

O D I T S Z A P C
* * * * * *

| Format | Examples | | Clocks |
|---|---|---|---|
| NEG reg | NEG AX<br>NEG CX<br>NEG EDX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |

| | | | |
|---|---|---|---|
| NEG mem | NEG DATA_UP<br>NEG NUMB<br>NEG WORD PTR [DI] | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

**NOP** No operation

10010000
Example

O D I T S Z A P C

| | | |
|---|---|---|
| NOP | | Clocks |
| | 8086 | 3 |
| | 8088 | 3 |
| | 80286 | 3 |
| | 80386 | 3 |
| | 80486 | 3 |
| | Pentium | 1 |

**NOT** One's complement

1111011w oo010mmm disp
Format

O D I T S Z A P C

| | | | |
|---|---|---|---|
| Examples | | | Clocks |
| NOT reg | NOT AX<br>NOT CX<br>NOT EDX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| NOT mem | NOT DATA_TRUE<br>NOT NUMB<br>NOT WORD PTR [DI] | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

## OR Inclusive-OR

000010dw oorrrmmm disp

O D I T S Z A P C
0 * * ? * 0

| Format | Examples | | Clocks |
|---|---|---|---|
| OR reg,reg | OR CL,BL<br>OR CX,DX<br>OR ECX,EBX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| OR mem,reg | OR DATAZ,CL<br>OR NUMB,CX<br>OR [DI],CX | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/2 |
| OR reg,mem | OR CL,DATA_R_US<br>OR CX,NUMB<br>OR CX,[SI] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

100000sw oo001mmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| OR reg,imm | OR CL,3<br>OR DX,1000H<br>OR EBX,100000H | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |

| OR mem,imm | OR DATA9,33<br>OR NUMB,4AH<br>OR NUMS,123498H<br>OR BYTE PTR [ECX],2 | 8086 | 17 + ea |
|---|---|---|---|
| | | 8088 | 25 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 0000110w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| OR acc,imm | OR AL,3<br>OR AX,1000H<br>OR EAX,100000H | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

**OUT** Output data to port

| 1110011w port number<br>Format | Examples | O D I T<br>Clocks | S Z A P C |
|---|---|---|---|
| OUT pt,acc | OUT 12H,AL<br>OUT 12H,AX<br>OUT 0FFH,AL<br>OUT 0FEH,AX<br>OUT 10H,EAX | 8086 | 10 |
| | | 8088 | 14 |
| | | 80286 | 3 |
| | | 80386 | 10 |
| | | 80486 | 10 |
| | | Pentium | 12 |

| 1110111w<br>Format | Examples | | Clocks |
|---|---|---|---|
| OUT DX,acc | OUT DX,AL<br>OUT DX,AX<br>OUT DX,EAX | 8086 | 8 |
| | | 8088 | 12 |
| | | 80286 | 3 |
| | | 80386 | 11 |
| | | 80486 | 10 |
| | | Pentium | 25 |

## OUTS Output string data to port

1110011w port number

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| OUTSB<br>OUTSW<br>OUTSD | OUTSB<br>OUTSW<br>OUTSD<br>OUTS DATAD<br>REP OUTSB | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 |
| | | 80386 | 14 |
| | | 80486 | 10 |
| | | Pentium | 13 |

## POP Pop data from stack

01011rrr

| Format | Examples | O D I T | S Z A P C Clocks |
|---|---|---|---|
| POP reg | POP CX<br>POP AX<br>POP EBX | 8086 | 8 |
| | | 8088 | 12 |
| | | 80286 | 5 |
| | | 80386 | 4 |
| | | 80486 | 1 |
| | | Pentium | 1 |

10001111 oo000mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| POP mem | POP DATAA<br>POP LISTS<br>POP NUMBS | 8086 | 17 + ea |
| | | 8088 | 25 + ea |
| | | 80286 | 5 |
| | | 80386 | 5 |
| | | 80486 | 4 |
| | | Pentium | 3 |

| 00sss111<br>Format | Examples | | Clocks |
|---|---|---|---|
| POP seg | POP DS<br>POP ES<br>POP SS | 8086 | 8 |
| | | 8088 | 12 |
| | | 80286 | 5 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 3 |

| 00001111 10sss001<br>Format | Examples | | Clocks |
|---|---|---|---|
| POP seg | POP FS<br>POP GS | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 3 |

**POPA/POPAD** Pop all registers from stack

| 01100001<br>Examples | O D I T S Z A P C<br>Clocks | |
|---|---|---|
| POPA<br>POPAD | 8086 | — |
| | 8088 | — |
| | 80286 | 19 |
| | 80386 | 24 |
| | 80486 | 9 |
| | Pentium | 5 |

## POPF/POPFD Pop flags from stack

10011101

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| * | * | * | * | * | * | * | * | * |

Examples

| | | |
|---|---|---|
| POPF<br>POPFD | 8086 | — |
| | 8088 | — |
| | 80286 | 5 |
| | 80386 | 5 |
| | 80486 | 6 |
| | Pentium | 4 |

## PUSH Push data onto stack

01010rrr

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|

| Format | Examples | | |
|---|---|---|---|
| PUSH reg | PUSH CX<br>PUSH AX<br>PUSH ECX | 8086 | 11 |
| | | 8088 | 15 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

11111111 oo110mmm disp

| Format | Examples | | |
|---|---|---|---|
| PUSH mem | PUSH DATAD<br>PUSH LISTS<br>PUSH NUMB<br>PUSH DWORD PTR [ECX] | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 5 |
| | | 80386 | 5 |
| | | 80486 | 4 |
| | | Pentium | 1/2 |

| 00sss110<br>Format | Examples | | Clocks |
|---|---|---|---|
| PUSH seg | PUSH DS<br>PUSH CS<br>PUSH ES | 8086 | 10 |
| | | 8088 | 14 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 3 |
| | | Pentium | 1 |

| 00001111 10sss000<br>Format | Examples | | Clocks |
|---|---|---|---|
| PUSH seg | PUSH FS<br>PUSH GS | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 2 |
| | | 80486 | 3 |
| | | Pentium | 1 |

| 011010s0 data<br>Format | Examples | | Clocks |
|---|---|---|---|
| PUSH imm | PUSH 2000H<br>PUSH 5322H<br>PUSHW 10H<br>PUSHD 100000H | 8086 | — |
| | | 8088 | — |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## PUSHA/PUSHAD — Push all registers

01100000
Example

O D I T S Z A P C
Clocks

PUSHA
PUSHAD

| | |
|---|---|
| 8086 | — |
| 8088 | — |
| 80286 | 17 |
| 80386 | 18 |
| 80486 | 11 |
| Pentium | 5 |

## PUSHF/PUSHFD — Push flags onto stack

10011100
Examples

O D I T S Z A P C
Clocks

PUSHF
PUSHFD

| | |
|---|---|
| 8086 | 10 |
| 8088 | 14 |
| 80286 | 3 |
| 80386 | 4 |
| 80486 | 3 |
| Pentium | 3 |

## RCL/RCR/ROL/ROR — Rotate

1101000w ooTTTmmm disp

O D I T S Z A P C
* *

TTT = 000 = ROL
TTT = 001 = ROR
TTT = 010 = RCL
TTT = 011 = RCR

| Format | Examples | | Clocks |
|---|---|---|---|
| ROL reg,1<br>ROR reg,1 | ROL CL,1<br>ROL DX,1<br>ROR CH,1<br>ROL SI,1 | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 2 |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| | | | |
|---|---|---|---|
| RCL reg,1<br>RCR reg,1 | RCL CL,1<br>RCL SI,1<br>RCR AH,1<br>RCR EBX,1 | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 2 |
| | | 80386 | 9 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |
| ROL mem,1<br>ROR mem,1 | ROL DATA4,1<br>ROL BYTE PTR [DI],1<br>ROR NUMB,1<br>ROR DWORD PTR [ECX],1 | 8086 | 15 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 4 |
| | | Pentium | 1/3 |
| RCL mem,1<br>RCR mem,1 | RCL DATA7,1<br>RCL BYTE PTR [DI],1<br>RCR NUMB,1<br>RCR WORD PTR [ECX],1 | 8086 | 15 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 10 |
| | | 80486 | 4 |
| | | Pentium | 1/3 |
| 1101001w ooTTTmmm disp<br>Format | Examples | | Clocks |
| ROL reg,CL<br>ROR reg,CL | ROL CH,CL<br>ROL DX,CL<br>ROR CH,CL<br>ROL SI,CL | 8086 | 8 + 4n |
| | | 8088 | 8 + 4n |
| | | 80286 | 5 + n |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4 |
| RCL reg,CL<br>RCR reg,CL | RCL DL,CL<br>RCL SI,CL<br>RCR AH,CL<br>RCR BX,CL | 8086 | 8 + 4n |
| | | 8088 | 8 + 4n |
| | | 80286 | 5 + n |
| | | 80386 | 9 |
| | | 80486 | 8 |
| | | Pentium | 4 |

| | | | |
|---|---|---|---|
| ROL mem,CL<br>ROR mem,CL | ROL DATA3,CL<br>ROL BYTE PTR [DI],CL<br>ROR NUMB,CL<br>ROR WORD PTR [ECX],CL | 8086 | 20 + a + 4n |
| | | 8088 | 28 + a + 4n |
| | | 80286 | 8 + n |
| | | 80386 | 7 |
| | | 80486 | 4 |
| | | Pentium | 4 |
| RCL mem,CL<br>RCR mem,CL | RCL DATA0,CL<br>RCL BYTE PTR [DI],CL<br>RCR NUMB,CL<br>RCR WORD PTR [ECX],CL | 8086 | 20 + a + 4n |
| | | 8088 | 28 + a + 4n |
| | | 80286 | 8 + n |
| | | 80386 | 10 |
| | | 80486 | 9 |
| | | Pentium | 4 |
| 1100000w ooTTTmmm disp data | | | |
| Format | Examples | | Clocks |
| ROL reg,imm<br>ROR reg,imm | ROL CL,4<br>ROL DX,5<br>ROR CH,12<br>ROL SI,9 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 + n |
| | | 80386 | 3 |
| | | 80486 | 2 |
| | | Pentium | 1/3 |
| RCL reg,imm<br>RCR reg,imm | RCL CL,2<br>RCL SI,3<br>RCR AH,5<br>RCR BX,13 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 + n |
| | | 80386 | 9 |
| | | 80486 | 8 |
| | | Pentium | 1/3 |

| | | | |
|---|---|---|---|
| ROL mem,imm<br>ROR mem,imm | ROL DATAH,4<br>ROL BYTE PTR [DI],2<br>ROR NUMB,2<br>ROR WORD PTR [ECX],3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 8 + n |
| | | 80386 | 7 |
| | | 80486 | 4 |
| | | Pentium | 1/3 |
| RCL mem,imm<br>RCR mem,imm | RCL DATAK,6<br>RCL BYTE PTR [DI],7<br>RCR NUMB,6<br>RCR WORD PTR [ECX],5 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 8 + n |
| | | 80386 | 10 |
| | | 80486 | 9 |
| | | Pentium | 1/3 |

**RDMSR** Read from model specific register

00001111 00110010
Example

O D I T S Z A P C
Clocks

| | | |
|---|---|---|
| RDMSR | 8086 | — |
| | 8088 | — |
| | 80286 | — |
| | 80386 | — |
| | 80486 | — |
| | Pentium | 20–24 |

**REP** Repeat prefix

11110011 1010010w
Format

Examples

O D I T S Z A P C
Clocks

| | | | |
|---|---|---|---|
| REP MOVS | REP MOVSB<br>REP MOVSW<br>REP MOVSD<br>REP MOVS DATA1,DATA2 | 8086 | 9 + 17n |
| | | 8088 | 9 + 25n |
| | | 80286 | 5 + 4n |
| | | 80386 | 8 + 4n |
| | | 80486 | 12 + 3n |
| | | Pentium | 13n |

| 11110011 1010101w<br>Format | Examples | | Clocks |
|---|---|---|---|
| REP STOS | REP STOSB<br>REP STOSW<br>REP STOSD<br>REP STOS DATA3 | 8086 | 9 + 10n |
| | | 8088 | 9 + 14n |
| | | 80286 | 4 + 3n |
| | | 80386 | 5 + 5n |
| | | 80486 | 7 + 4n |
| | | Pentium | 9n |

| 11110011 0110110w<br>Format | Examples | | Clocks |
|---|---|---|---|
| REP INS | REP INSB<br>REP INSW<br>REP INSD<br>REP INS DATA4 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 + 4n |
| | | 80386 | 13 + 6n |
| | | 80486 | 16 + 8n |
| | | Pentium | 8 + 3n |

| 11110011 0110111w<br>Format | Examples | | Clocks |
|---|---|---|---|
| REP OUTS | REP OUTSB<br>REP OUTSW<br>REP OUTSD<br>REP OUTS DATA5 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 + 4n |
| | | 80386 | 12 + 5n |
| | | 80486 | 17 + 5n |
| | | Pentium | 13 + 4n |

**REPE/REPNE** Repeat conditional

| 11110011 1010011w<br>Format | Examples | O D I T S Z A P C<br>*<br>Clocks | |
|---|---|---|---|
| REPE CMPS | REPE CMPSB<br>REPE CMPSW<br>REPE CMPSD<br>REPE CMPS DATA6,DATA7 | 8086 | 9 + 22n |
| | | 8088 | 9 + 30n |
| | | 80286 | 5 + 9n |
| | | 80386 | 5 + 9n |
| | | 80486 | 7 + 7n |
| | | Pentium | 8 + 4n |

| 11110011 1010111w<br>Format | Examples | | Clocks |
|---|---|---|---|
| REPE SCAS | REPE SCASB<br>REPE SCASW<br>REPE SCASD<br>REPE SCAS DATA8 | 8086 | 9 + 15n |
| | | 8088 | 9 + 19n |
| | | 80286 | 5 + 8n |
| | | 80386 | 5 + 8n |
| | | 80486 | 7 + 5n |
| | | Pentium | 9 + 4n |

| 11110010 1010011w<br>Format | Examples | | Clocks |
|---|---|---|---|
| REPNE CMPS | REPNE CMPSB<br>REPNE CMPSW<br>REPNE CMPSD<br>REPNE CMPS DATA9,DATA10 | 8086 | 9 + 22n |
| | | 8088 | 9 + 30n |
| | | 80286 | 5 + 9n |
| | | 80386 | 5 + 9n |
| | | 80486 | 7 + 7n |
| | | Pentium | 8 + 4n |

| 11110010 1010111w<br>Format | Examples | | Clocks |
|---|---|---|---|
| REPNE SCAS | REPNE SCASB<br>REPNE SCASW<br>REPNE SCASD<br>REPNE SCAS DATA11 | 8086 | 9 + 15n |
| | | 8088 | 9 + 19n |
| | | 80286 | 5 + 8n |
| | | 80386 | 5 + 8n |
| | | 80486 | 7 + 5n |
| | | Pentium | 9 + 4n |

**RET** Return from procedure

| 11000011<br>Example | O D I T | S Z A P C<br>Clocks |
|---|---|---|
| RET<br>(near) | 8086 | 16 |
| | 8088 | 20 |
| | 80286 | 11 |
| | 80386 | 10 |
| | 80486 | 5 |
| | Pentium | 2 |

| 11000010 data Format | Examples | | Clocks |
|---|---|---|---|
| RET imm (near) | RET 4<br>RET 100H | 8086 | 20 |
| | | 8088 | 24 |
| | | 80286 | 11 |
| | | 80386 | 10 |
| | | 80486 | 5 |
| | | Pentium | 3 |

| 11001011 Example | | Clocks |
|---|---|---|
| RET (far) | 8086 | 26 |
| | 8088 | 34 |
| | 80286 | 15 |
| | 80386 | 18 |
| | 80486 | 13 |
| | Pentium | 23 |

| 11001010 data Format | Examples | | Clocks |
|---|---|---|---|
| RET imm (far) | RET 4<br>RET 100H | 8086 | 25 |
| | | 8088 | 33 |
| | | 80286 | 11 |
| | | 80386 | 10 |
| | | 80486 | 5 |
| | | Pentium | 23 |

## RSM — Resume from system management mode

00001111 10101010

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| * | * | * | * | * | * | * | * | * |

Example

| RSM | Clocks |
|---|---|
| 8086 | — |
| 8088 | — |
| 80286 | — |
| 80386 | — |
| 80486 | — |
| Pentium | 83 |

## SAHF — Store AH into flags

10011110

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| | | | | * | * | * | * | * |

Example

| SAHF | Clocks |
|---|---|
| 8086 | 4 |
| 8088 | 4 |
| 80286 | 2 |
| 80386 | 3 |
| 80486 | 2 |
| Pentium | 2 |

## SAL/SAR/SHL/SHR — Shift

1101000w ooTTTmmm disp

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| * | | | | * | * | ? | * | * |

TTT = 100 = SHL/SAL
TTT = 101 = SHR
TTT = 111 = SAR

| Format | Examples | | Clocks |
|---|---|---|---|
| SAL reg,1<br>SHL reg,1<br>SHR reg,1<br>SAR reg,1 | SAL CL,1<br>SHL DX,1<br>SHR CH,1<br>SAR SI,1 | 8086 | 2 |
| | | 8088 | 2 |
| | | 80286 | 2 |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| | | | |
|---|---|---|---|
| SAL mem,1<br>SHL mem,1<br>SHR mem,1<br>SAR mem,1 | SAL DATAR,1<br>SHL BYTE PTR [DI],1<br>SHR NUMB,1<br>SAR WORD PTR [ECX],1 | 8086 | 15 + ea |
| | | 8088 | 23 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 4 |
| | | Pentium | 1/3 |

1101001w ooTTTmmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| SAL reg,CL<br>SHL reg,CL<br>SHR reg,CL<br>SAR reg,CL | SAL CH,CL<br>SHL DX,CL<br>SHR CH,CL<br>SAR SI,CL | 8086 | 8 + 4n |
| | | 8088 | 8 + 4n |
| | | 80286 | 5 + n |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4 |
| SAL mem,CL<br>SHL mem,CL<br>SHR mem,CL<br>SAR mem,CL | SAL DATAU,CL<br>SHL BYTE PTR [DI],CL<br>SHR NUMB,CL<br>SAR WORD PTR [ECX],CL | 8086 | 20 + a + 4n |
| | | 8088 | 28 + a + 4n |
| | | 80286 | 8 + n |
| | | 80386 | 7 |
| | | 80486 | 4 |
| | | Pentium | 4 |

1100000w ooTTTmmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| SAL reg,imm<br>SHL reg,imm<br>SHR reg,imm<br>SAR reg,imm | SAL CL,4<br>SHL DX,5<br>SHR CH,12<br>SAR SI,9 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 5 + n |
| | | 80386 | 3 |
| | | 80486 | 2 |
| | | Pentium | 1/3 |

| | | | |
|---|---|---|---|
| SAL mem,imm<br>SHL mem,imm<br>SHR mem,imm<br>SAR mem,imm | SAL DATAP,6<br>SHL BYTE PTR [DI],7<br>SHR NUMB,6<br>SAR WORD PTR [ECX],5 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 8 + n |
| | | 80386 | 7 |
| | | 80486 | 4 |
| | | Pentium | 1/3 |

## SBB Subtract with borrow

000110dw oorrrmmm disp

O D I T S Z A P C
* * * * * * *

| Format | Examples | | Clocks |
|---|---|---|---|
| SBB reg,reg | SBB CL,DL<br>SBB AX,DX<br>SBB CH,CL<br>SBB EAX,EBX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| SBB mem,reg | SBB DATAJ,CL<br>SBB BYTES,CX<br>SBB NUMBS,ECX<br>SBB [EAX],CX | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |
| SBB reg,mem | SBB CL,DATAL<br>SBB CX,BYTES<br>SBB ECX,NUMBS<br>SBB CX,[EDX] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 2 |
| | | Pentium | 1/3 |

100000sw oo011mmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| SBB reg,imm | SBB CL,4<br>SBB DX,5<br>SBB CH,12<br>SBB SI,9 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| SBB mem,imm | SBB DATAF,6<br>SBB BYTE PTR [DI],7<br>SBB NUMB,6<br>SBB WORD PTR [ECX],5 | 8086 | 17 + ea |
| | | 8088 | 25 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

0001110w data

| Format | Examples | | Clocks |
|---|---|---|---|
| SBB acc,imm | SBB AL,4<br>SBB AX,5<br>SBB AH,12<br>SBB AX,9 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

**SCAS** Scan string

1010111w

| | | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|
| | | * | | | | * | * | * | * | * |

| Format | Examples | | Clocks |
|---|---|---|---|
| SCASB<br>SCASW<br>SCASD | SCASB<br>SCASW<br>SCASD<br>SCAS DATAH<br>REP SCASB | 8086 | 15 |
| | | 8088 | 19 |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 6 |
| | | Pentium | 4 |

## SET Set on condition

00001111 1001cccc oo000mmm

| Format | Examples | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Clocks | | | | | | | | |
| SETcd reg8 | SETA BL<br>SETB CH<br>SETG DL<br>SETE BH<br>SETZ AL | 8086 | | | | — | | | | |
| | | 8088 | | | | — | | | | |
| | | 80286 | | | | — | | | | |
| | | 80386 | | | | 4 | | | | |
| | | 80486 | | | | 3 | | | | |
| | | Pentium | | | | 1/2 | | | | |
| SETcd mem8 | SETE DATAK<br>SETLE BYTES | 8086 | | | | — | | | | |
| | | 8088 | | | | — | | | | |
| | | 80286 | | | | — | | | | |
| | | 80386 | | | | 5 | | | | |
| | | 80486 | | | | 3 | | | | |
| | | Pentium | | | | 1/2 | | | | |

| Condition Codes | Mnemonic | Flag | Description |
|---|---|---|---|
| 0000 | SETO | O = 1 | Set if overflow |
| 0001 | SETNO | O = 0 | Set if no overflow |
| 0010 | SETB/SETNAE | C = 1 | Set if below |
| 0011 | SETAE/SETNB | C = 0 | Set if above or equal |
| 0100 | SETE/SETZ | Z = 1 | Set if equal/zero |
| 0101 | SETNE/SETNZ | Z = 0 | Set if not equal/not zero |
| 0110 | SETBE/SETNA | C = 1 + Z = 1 | Set if below or equal |
| 0111 | SETA/SETNBE | C = 0 • Z = 0 | Set if above |
| 1000 | SETS | S = 1 | Set if sign |
| 1001 | SETNS | S = 0 | Set if no sign |
| 1010 | SETP/SETPE | P = 1 | Set if parity even |
| 1011 | SETNP/SETPO | P = 0 | Set if parity odd |
| 1100 | SETL/SETNGE | S • O | Set if less than |
| 1101 | SETGE/SETNL | S = O | Set if greater or equal |
| 1110 | SETLE/SETNG | Z = 1 + S • O | Set if less than or equal |
| 1111 | SETG/SETNLE | Z = 0 + S = O | Set if greater |

## SGDT/SIDT/SLDT Store descriptor table

| 00001111 00000001 oo000mmm disp<br>Format | Examples | O D I T S Z A P C | Clocks |
|---|---|---|---|
| SGDT mem | SGDT MEMORY<br>SGDT GLOBAL | 8086 | — |
| | | 8088 | — |
| | | 80286 | 11 |
| | | 80386 | 9 |
| | | 80486 | 10 |
| | | Pentium | 4 |

| 00001111 00000001 oo001mmm disp<br>Format | Examples | | Clocks |
|---|---|---|---|
| SIDT mem | SIDT DATAS<br>SIDT INTERRUPT | 8086 | — |
| | | 8088 | — |
| | | 80286 | 12 |
| | | 80386 | 9 |
| | | 80486 | 10 |
| | | Pentium | 4 |

| 00001111 00000000 oo000mmm disp<br>Format | Examples | | Clocks |
|---|---|---|---|
| SLDT reg | SLDT CX<br>SLDT DX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 2 |
| | | Pentium | 2 |
| SLDT mem | SLDT NUMBS<br>SLDT LOCALS | 8086 | — |
| | | 8088 | — |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 3 |
| | | Pentium | 2 |

## SHLD/SHRD Double precision shift

00001111 10100100 oorrrmmm disp data

O D I T S Z A P C
? * * ? * *

| Format | Examples | | Clocks |
|---|---|---|---|
| SHLD reg,reg,imm | SHLD AX,CX,10<br>SHLD DX,BX,8<br>SHLD CX,DX,2 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 3 |
| | | 80486 | 2 |
| | | Pentium | 4 |
| SHLDSHLD mem,reg,imm | DATAQ,CX,8 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 4 |

00001111 10101100 oorrrmmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| SHRD reg,reg,imm | SHRD CX,DX,2 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 3 |
| | | 80486 | 2 |
| | | Pentium | 4 |
| SHRD mem,reg,imm | SHRD DATAT,CX,3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 4 |

| 00001111 10100101 oorrrmmm disp<br>Format | Examples | | Clocks |
|---|---|---|---|
| SHLD<br>reg,reg,CL | SHLD DX,BX,CL | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4/5 |
| SHLD<br>mem,reg,CL | SHLD DATA4,AX,CL | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 4/5 |

| 00001111 10101101 oorrrmmm disp<br>Format | Examples | | Clocks |
|---|---|---|---|
| SHRD<br>mem,reg,CL | SHLD DX,BX,CL | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4/5 |
| SHRD<br>mem,reg,CL | SHRD DATA3,AX,CL | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 4/5 |

## SMSW Store machine status word

00001111 00000001 oo100mmm disp
(should only be used by the 80286)

O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| SMSW reg | SMSW AX<br>SMSW DX<br>SMSW CX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 2 |
| | | 80386 | 10 |
| | | 80486 | 2 |
| | | Pentium | 4 |
| SMSW mem | SMSW DATAZ1 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 3 |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 4 |

## STC Set carry flag

11111001

O D I T S Z A P C
1

| Example | | Clocks |
|---|---|---|
| STC | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 2 |
| | Pentium | 2 |

**STD** Set direction flag

11111101 O D I T S Z A P C
1

Example

| | | |
|---|---|---|
| STD | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 2 |
| | 80486 | 2 |
| | Pentium | 2 |

**STI** Set interrupt flag

11111011 O D I T S Z A P C
1

Example

| | | Clocks |
|---|---|---|
| STI | 8086 | 2 |
| | 8088 | 2 |
| | 80286 | 2 |
| | 80386 | 3 |
| | 80486 | 5 |
| | Pentium | 7 |

**STOS** Store string data

1010101w O D I T S Z A P C

| Format | Examples | | Clocks |
|---|---|---|---|
| STOSB<br>STOSW<br>STOSD | STOSB<br>STOSW<br>STOSD<br>STOS DATA4<br>REP STOSB | 8086 | 11 |
| | | 8088 | 15 |
| | | 80286 | 3 |
| | | 80386 | 4 |
| | | 80486 | 5 |
| | | Pentium | 3 |

## STR Store task register

00001111 00000000 oo001mmm disp

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

| Format | Examples | | Clocks |
|---|---|---|---|
| STR reg | STR DX<br>STR CX<br>STR AX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 2 |
| | | Pentium | 2 |
| STR mem | STR DATA3 | 8086 | — |
| | | 8088 | — |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 3 |
| | | Pentium | 2 |

## SUB Subtract

001010dw oorrrmmm disp

| O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|
| * | | | | * | * | * | * | * |

| Format | Examples | | Clocks |
|---|---|---|---|
| SUB reg,reg | SUB CL,DL<br>SUB AX,DX<br>SUB CH,CL<br>SUB EAX,EBX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| SUB mem,reg | SUB BEAVUS,CL<br>SUB BYTES,CX<br>SUB NUMBS,ECX<br>SUB [EAX],CX | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| SUB reg,mem | SUB CL,BUTHEAD<br>SUB CX,BYTES<br>SUB ECX,NUMBS<br>SUB CX,[EDX] | 8086 | 9 + ea |
|---|---|---|---|
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

| 100000sw oo101mmm disp data<br>Format | Examples | | Clocks |
|---|---|---|---|
| SUB reg,imm | SUB CL,4<br>SUB DX,5<br>SUB CH,12<br>SUB SI,9 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |
| SUB mem,imm | SUB MTV,6<br>SUB BYTE PTR [DI],7<br>SUB NUMB,6<br>SUB WORD PTR [ECX],5 | 8086 | 17 + ea |
| | | 8088 | 25 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 0010110w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| SUB acc,imm | SUB AL,4<br>SUB AX,5<br>SUB AH,12<br>SUB AX,9 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## TEST Test operands (logical compare)

1000011w oorrrmmm disp

O D I T S Z A P C
0 * * ? * 0

| Format | Examples | | Clocks |
|---|---|---|---|
| TEST reg,reg | TEST CL,DL<br>TEST CX,DX<br>TEST CL,CH<br>TEST ECX,EBX | 8086 | 5 |
| | | 8088 | 5 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| TEST reg,mem<br>mem,reg | TEST DATA1,CL<br>TEST CL,DATA1 | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 6 |
| | | 80386 | 5 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

1111011w oo000mmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| TEST reg,imm | TEST CL,4<br>TEST DX,5<br>TEST CH,12H<br>TEST SI,256 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| TEST mem,imm | TESTDATA3,6 | 8086 | 11 + ea |
| | | 8088 | 11 + ea |
| | | 80286 | 6 |
| | | 80386 | 5 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

| 1010100w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| TEST acc,imm | TEST AL,4<br>TEST AX,5<br>TEST AH,12<br>TEST AX,9<br>TEST EAX,2 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

## VERR/VERW Verify read or write

00001111 00000000 oo100mmm disp

O D I T S Z A P C
*

| Format | Examples | | Clocks |
|---|---|---|---|
| VERR reg | VERR BX<br>VERR CX<br>VERR DX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 14 |
| | | 80386 | 10 |
| | | 80486 | 11 |
| | | Pentium | 7 |
| VERR mem | VERR DATA_SEVEN | 8086 | — |
| | | 8088 | — |
| | | 80286 | 16 |
| | | 80386 | 11 |
| | | 80486 | 11 |
| | | Pentium | 7 |

00001111 00000000 oo101mmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| VERW reg | VERW AX<br>VERW CX<br>VERW DX | 8086 | — |
| | | 8088 | — |
| | | 80286 | 14 |
| | | 80386 | 15 |
| | | 80486 | 11 |
| | | Pentium | 7 |

| | | | |
|---|---|---|---|
| VERW mem | VERW DATA_SEG | 8086 | — |
| | | 8088 | — |
| | | 80286 | 16 |
| | | 80386 | 16 |
| | | 80486 | 11 |
| | | Pentium | 7 |

**WAIT** Wait for coprocessor

| 10011011<br>Examples | O D I T S Z A P C<br>Clocks | |
|---|---|---|
| WAIT<br>FWAIT | 8086 | 4 |
| | 8088 | 4 |
| | 80286 | 3 |
| | 80386 | 6 |
| | 80486 | 6 |
| | Pentium | 1 |

**WBINVD** Write-back and invalidate data cache

| 00001111 00001001<br>Example | O D I T S Z A P C<br>Clocks | |
|---|---|---|
| WBINVD | 8086 | — |
| | 8088 | — |
| | 80286 | — |
| | 80386 | — |
| | 80486 | 5 |
| | Pentium | 2000+ |

## WRMSR Write to model specific register

00001111 00110000

Example

| | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

WRMSR

| | Clocks |
|---|---|
| 8086 | — |
| 8088 | — |
| 80286 | — |
| 80386 | — |
| 80486 | — |
| Pentium | 30–45 |

## XADD Exchange and add

00001111 1100000w 11rrrrrr

| | O | D | I | T | S | Z | A | P | C |
|---|---|---|---|---|---|---|---|---|---|
| | * | | | | * | * | * | * | * |

| Format | Examples | | Clocks |
|---|---|---|---|
| XADD reg,reg | XADD EBX,ECX<br>XADD EDX,EAX<br>XADD EDI,EBP | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | — |
| | | 80486 | 3 |
| | | Pentium | 3/4 |

00001111 1100000w oorrrmmm disp

| Format | Examples | | Clocks |
|---|---|---|---|
| XADD mem,reg | XADD DATA5,EAX<br>XADD [DI],EAX<br>XADD [ECX],EDX | 8086 | — |
| | | 8088 | — |
| | | 80286 | — |
| | | 80386 | — |
| | | 80486 | 4 |
| | | Pentium | 3/4 |

## XCHG Exchange

1000011w 1oorrrmmm

| Format | Examples | O D I T S Z A P C Clocks | |
|---|---|---|---|
| XCHG reg,reg | XCHG BL,CL<br>XCHG AX,DX<br>XCHG EDI,EBP | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 3 |
| XCHG reg,mem<br>mem,reg | XCHG CL,DATA4<br>XCHG DATA7,CL<br>XCHG DX,[DI]<br>XCHG ECX,[EBP] | 8086 | 17 + ea |
| | | 8088 | 25 + ea |
| | | 80286 | 5 |
| | | 80386 | 5 |
| | | 80486 | 5 |
| | | Pentium | 3 |

10010reg

| Format | Examples | | Clocks |
|---|---|---|---|
| XCHG acc,reg<br>XCHG reg,acc | XCHG AL,CL<br>XCHG DX,AX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 3 |
| | | 80386 | 3 |
| | | 80486 | 3 |
| | | Pentium | 2 |

## XLAT Translate

11010111

| Example | O D I T S Z A P C Clocks | |
|---|---|---|
| XLAT | 8086 | 11 |
| | 8088 | 11 |
| | 80286 | 5 |
| | 80386 | 3 |
| | 80486 | 4 |
| | Pentium | 4 |

## XOR Exclusive-OR

001100dw oorrrmmm disp

O D I T S Z A P C
0 * * ? * 0

| Format | Examples | | Clocks |
|---|---|---|---|
| XOR reg,reg | XOR BL,CL<br>XOR CX,DX<br>XOR CH,CL<br>XOR EAX,EBX | 8086 | 3 |
| | | 8088 | 3 |
| | | 80286 | 2 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/2 |
| XOR mem,reg | XOR DATAD,CL<br>XOR BYTES,CX<br>XOR NUMBS,ECX<br>XOR [EAX],CX | 8086 | 16 + ea |
| | | 8088 | 24 + ea |
| | | 80286 | 7 |
| | | 80386 | 6 |
| | | 80486 | 3 |
| | | Pentium | 1/2 |
| XOR reg,mem | XOR CL,DATAB<br>XOR CX,BYTES<br>XOR ECX,NUMBS<br>XOR CX,[EDX] | 8086 | 9 + ea |
| | | 8088 | 13 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 2 |
| | | Pentium | 1/2 |

100000sw oo110mmm disp data

| Format | Examples | | Clocks |
|---|---|---|---|
| XOR reg,imm | XOR BL,33<br>XOR CX,234H<br>XOR CH,'A'<br>XOR EAX,123445 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1/3 |

| XOR mem,imm | XOR DATA_NEW,34<br>XOR BYTES,1234<br>XOR NUMBS,123<br>XOR [EAX],11 | 8086 | 17 + ea |
|---|---|---|---|
| | | 8088 | 25 + ea |
| | | 80286 | 7 |
| | | 80386 | 7 |
| | | 80486 | 3 |
| | | Pentium | 1/3 |

| 0011010w data<br>Format | Examples | | Clocks |
|---|---|---|---|
| XOR acc,imm | XOR AL,33<br>XOR AX,234H<br>XOR AL,'A'<br>XOR EAX,123445 | 8086 | 4 |
| | | 8088 | 4 |
| | | 80286 | 3 |
| | | 80386 | 2 |
| | | 80486 | 1 |
| | | Pentium | 1 |

# APPENDIX C

# Flag Bit Changes

This appendix shows only the instructions that actually change the flag bits. Any instruction not listed does not affect any of the flag bits.

| *Instruction* | *Flags* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | O | D | I | T | S | Z | A | P | C |
| AAA | ? | | | ? | ? | * | ? | * | |
| AAD | ? | | | * | * | ? | * | ? | |
| AAM | ? | | | * | * | ? | * | ? | |
| AAS | ? | | | ? | ? | * | ? | * | |
| ADC | * | | | * | * | * | * | * | |
| ADD | * | | | * | * | * | * | * | |
| AND | 0 | | | * | * | ? | * | 0 | |
| ARPL | | | | | * | | | | |
| BSF | | | | | * | | | | |
| BSR | | | | | * | | | | |
| BT | | | | | | | | | * |
| BTC | | | | | | | | | * |
| BTR | | | | | | | | | * |
| BTS | | | | | | | | | * |
| CLC | | | | | | | | | 0 |
| CLD | | 0 | | | | | | | |
| CLI | | | 0 | | | | | | |
| CMC | | | | | | | | | * |
| CMP | * | | | | * | * | * | * | * |
| CMPS | * | | | | * | * | * | * | * |

| *Instruction* | *Flags* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | O | D | I | T | S | Z | A | P | C |
| CMPXCHG | * | | | | * | * | * | * | * |
| CMPXCHG8B | | | | | | * | | | |
| DAA | ? | | | | * | * | * | * | * |
| DAS | ? | | | | * | * | * | * | * |
| DEC | * | | | | * | * | * | * | |
| DIV | ? | | | | ? | ? | ? | ? | ? |
| IDIV | ? | | | | ? | ? | ? | ? | ? |
| IMUL | * | | | | ? | ? | ? | ? | * |
| INC | * | | | | * | * | * | * | |
| IRET | * | * | * | * | * | * | * | * | * |
| LAR | | | | | * | | | | |
| LSL | | | | | * | | | | |
| MUL | * | | | | ? | ? | ? | ? | * |
| NEG | * | | | | * | * | * | * | * |
| OR | 0 | | | | * | * | ? | * | 0 |
| POPF/POPFD | * | * | * | * | * | * | * | * | * |
| RCL/RCR | * | | | | | | | | * |
| REPE/REPNE | | | | | | * | | | |
| ROL/ROR | * | | | | | | | | * |
| SAHF | | | | | * | * | * | * | * |
| SAL/SAR | * | | | | * | * | ? | * | * |
| SHL/SHR | * | | | | * | * | ? | * | * |
| SBB | * | | | | * | * | * | * | * |
| SCAS | * | | | | * | * | * | * | * |
| SHLD/SHRD | ? | | | | * | * | ? | * | * |
| STC | | | | | | | | | 1 |
| STD | | 1 | | | | | | | |
| STI | | | 1 | | | | | | |
| SUB | * | | | | * | * | * | * | * |
| TEST | 0 | | | | * | * | ? | * | 0 |
| VERR/VERW | | | | | * | | | | |
| XADD | * | | | | * | * | * | * | * |
| XOR | 0 | | | | * | * | ? | * | 0 |

# APPENDIX D

# Answers to Selected Even-Numbered Questions and Problems

## CHAPTER 1

2. Herman Hollerith
4. Knorad Zuse
6. ENIAC
8. 8080
10. 8086
12. 4G bytes
14. Millions of instructions per second
16. 1 or 0
18. 1,024K bytes
20. 1,000,000
22. 640K bytes
24. 1M bytes
26. 80386DX, 80486, and Pentium
28. The BIOS is the basic I/O system.
30. The XT system is a computer system based on the 8088 or 8086 microprocessor, while the AT is based on the 80286, 80386, 80486, or Pentium.
32. 8- or 16-bit
34. The EMS (expanded memory system) is memory that is added to the microprocessor in 64K sections, or pages, located within the system area of memory.
36. A TSR (terminate and stay resident) program is one that remains in the memory and can be activated by an interrupt or special keying sequence.
38. The CONFIG.SYS file sets up or configures the personal computer.
40. The COMMAND.COM program processes keyboard information for access to the DOS functions such as DIR (directory).
42. High memory is located at 100000H–10FFEFH.
44. Upper memory is usually located between memory locations C8000H and EFFFFH.
46. Refer to Figure 1–1.
48. Address, data, and control

50. $\overline{\text{MRDC}}$
52. Memory read
54. (a) 46H 52H 4EH 47H
    (b) 41H 72H 63H
    (c) 59H 61H 74H 65H 72H 2EH
    (d) 59H 65H 6CH 6CH
56. MESSAGE DB 'What time is it?'
58. (a) 0000 0011 1110 1000
    (b) 1111 1111 1000 1000
    (c) 0000 0011 0010 0000
    (d) 1111 0011 0111 0100
60. (a) 34 12
    (b) 11 A1
    (c) 00 B1
62. NUMBER DW 123AH
64. (a) –128
    (b) +51
    (c) –110
    (d) –119
66. (a) 0 01111111 10000000000000000000000
    (b) 1 10000010 01010100000000000000000
    (c) 0 10000101 00100001000000000000000
    (d) 1 10001001 00101100000000000000000

## CHAPTER 2

2. 16
4. EBX
6. The IP/EIP register is used in conjunction with CS to address the next sequential instruction in a program.
8. No, because the result of FFH (–1) plus 01H (+1) is zero (00H), a valid result.
10. The interrupt flag controls the INTR pin.
12. The segment register addresses the start of a 64K-byte segment of memory in the real mode.
14. (a) 12000H, (b) 21000H, (c) 24A00H, (d) 25000H, and (e) 3F12DH
16. DI
18. SS plus SP/ESP
20. (a) 12000H, (b) 21002H, (c) 26200H, (d) A1000H, and (e) 2CA00H
22. Memory locations 000000H through FFFFFFH, the real and extended memory systems.
24. The segment register is used in the protected mode to select a descriptor from either the local or global descriptor tables and also to request a privilege level for the access.
26. A00000H through A01000H
28. 00280000H through 00290000H
30. 3
32. 64K bytes

34. Bytes 0 and 1 = 3000H, bytes 2 and 3 = 0000H, bytes 4 and 5 = 9200H, and bytes 6 and 7 = 0380H.
36. The descriptor table is addressed by the descriptor table address register, an invisible register.
38. The program invisible registers are used during protected-mode operation to manage the memory system.

## CHAPTER 3

2. AH, AL, BH, BL, CH, CL, DH, and DL
4. EAX, EBX, ECX, EDX, ESP, EBP, EDI, and ESI
6. You may never mix register sizes.
8. (a) MOV EDX,EBX, (b) MOV CL,BL, (c) MOV BX,SI, (d) MOV DS,AX, and (e) MOV AH,AL
10. The # symbol is used by a few, but not most, assemblers to denote immediate data.
12. The [ ] symbols denote indirect data addressing.
14. Memory-to-memory data transfers are not allowed.
16. MOV WORD PTR [DI],1
18. The MOV BX,DATA instruction copies the contents of memory location DATA into BX, while MOV BX,OFFSET DATA copies the address of data into BX.
20. Nothing is wrong with this instruction. This form [BX][SI] must be used with the Intel ASM assembler program and can also be used by the MASM assembler program from Microsoft in lieu of [BX+SI].
22. (a) 11750H, (b) 11950H, and (c) 11700H
24. BP/EBP
26.
```
FIELDS      STRUC

F1          DW  ?
F2          DW  ?
F3          DW  ?
F4          DW  ?
F5          DW  ?

FIELDS      ENDS
```
28. Short, near, and far
30. The intersegment jump allows a jump to any location in the memory system, while the intrasegment jump allows a jump to any location in the current code segment.
32. 32
34. Short
36. JMP [BX]
38. The PUSH instruction stores 2 bytes and the PUSHD instruction stores 4 bytes.
40. AX, CX, DX, BX, SP, BP, DI, and SI
42. PUSHFD

## CHAPTER 4

2. The D bit specifies the direction of flow (from the register field to the R/M field, or vice versa), and the W bit specifies a word/doubleword or a byte.

4. DL
6. DS:[BX+DI] without a displacement
8. MOV AX,[BX]
10. 8B7702
12. You may not change the contents of the CS register.
14. 16
16. AX, CX, DX, BX, SP, BP, DI, and SI
18. (a) Pushes the 16-bit contents of AX onto the stack
    (b) Removes a 32-bit number from the stack and places it into ESI
    (c) Pushes the 16-bit contents of the data segment memory location addressed by BX onto the stack
    (d) Pushes the EFLAG register onto the stack
    (e) Removes a 16-bit number from the stack and places it into DS
    (f) Pushes a 32-bit number 4 onto the stack
20. The contents of EAX are placed on the stack at memory locations 02300H through 02303H.
22. One possible combination is SS = 0200H and SP = 0200H.
24. Both instructions load SI with the address of NUMB. The difference is that the LEA requires more time to execute than the MOV with an OFFSET instruction.
26. The LDS BX,NUMB instruction loads DS with the word contents of memory locations NUMB and BX with the word contents of NUMB+2.
28.
```
MOV   BX,NUMB
MOV   DX,BX
MOV   SI,BX
```
30. The CLD instruction clears the direction flag and the STD instruction sets it.
32. The LODSB instruction loads AL with the byte from the extra segment memory location addressed by DI. After the data are transferred, the contents of DI are incremented or decremented by 1, depending on the setting of the direction flag.
34. The OUTSB instruction transfers a copy of the data segment memory location addressed by SI to the I/O port whose address is located in DX. After the data are transferred, the contents of SI are either incremented or decremented depending on the setting of the direction flag.
36.
```
MOV   SI,OFFSET SOURCE
MOV   DI,OFFSET DEST
MOV   CX,12
REP   MOVSB
```
38. XCHG EBX,ESI
40. The XLAT instruction adds the contents of AL to BX to generate an address. The address is used to access memory data that are loaded into AL in a table.
42. The IN AL,12H instruction copies a byte from I/O port address 0012H into the AL register.
44. The segment override prefix allows the default segment to be changed to any other segment. For example, the MOV AL,ES:[BX] instruction transfers data from the extra segment because of the segment override prefix ES:
46.
```
XCHGAX,BX
XCHGECX,EDX
XHCGSI,DI
```

48. The DB directive allows bytes to be defined, DW allows words to be defined, and DD allows doublewords to be defined.
50. The EQU (equate) directive allows labels to be equated to values or other labels.
52. The .MODEL directive selects a memory model for the program.
54. Full segment definitions
56. The PROC directive indicates the start of a procedure and the ENDP directive indicates the end.
58. The USE16 statement causes the assembler to use the 16-bit instruction mode.
60.
```
DOIT  PROC  FAR
      MOV   AX,CS:DATA1
      MOV   BX,AX
      MOV   CX,AX
      MOV   DX,AX
      MOV   SI,AX
      RET

DOIT  ENDP
```

# CHAPTER 5

2. In this problem, sizes are mixed and the contents of AX are transferred to ECX, which are not allowed.
4. AX (sum) = 3100H, C = 0, A = 1, S = 0, Z = 0, and O = 0.
6.
```
ADD   AX,BX
ADD   AX,CX
ADD   AX,SP
MOV   DI,AX
```
8. ADC DX,BX
10. The assembler cannot tell whether the memory location addressed by BX is a byte, word, or doubleword.
12. DL = 81H, C = 0, A = 0, S = 1, P = 1, Z = 0, and O = 0.
14. DEC EBX
16. Both instructions subtract the source from the destination and change the flag bits to reflect the difference. The difference is that the CMP does not change the destination.
18. DX contains the most significant and AX contains the least significant part of the product.
20. EDX and EAX
22.
```
MOV   DL,5
MOV   AL,DL
MUL   DL
MUL   DL
```
24. AL
26. Division by zero and divide overflows cause a divide error interrupt.
28. AH
30. DAA and DAS
32. The AAM instruction multiplies the AX register by ten, leaving the quotient in AH and the remainder in AL.

34.
```
XCHG   AX,BX
ADD    AL,DL
DAA
MOV    DL,AL
MOV    AL,AH
ADC    AL,DH
DAA
MOV    DH,DL
XCHG   AX,BX
ADC    AL,CL
DAA
MOV    CL,AL
MOV    AL,AH
ADC    AL,CH
DAA
MOV    CH,AL
```

36.
```
AND    DH,1FH
MOV    BH,DH
```

38.
```
OR     DI,1FH
MOV    SI,DI
```

40.
```
OR     AX,000FH
AND    AX,1FFFH
XOR    AX,0380H
```

42. TEST CH,4

44. (a) SHR DI,3, (b) SHL AL,1, (c) ROL AL,3, (d) RCR EDX,1, and (e) SAR DH,1

46. Extra

48. The REPE (repeat while equal) prefix repeats the SCASB (compare AL with memory), while the comparison yields an equal condition. Note that for each repeated SCASB, the contents of CX are decremented. If CX becomes a zero, the repeated operation ends, even though the condition generated by the SCASB instruction may be equal.

50. The CMPSB instruction compares the data segment memory byte addressed by SI with the extra segment memory byte addressed by DI. It also increments/decrements the contents of SI and DI by one.

## CHAPTER 6

2. The near jump, as in JMP NEAR PTR LOCAL
4. The far jump, as in JMP FAR PTR DISTANT
6. (a) near, (b) short, and (c) far
8. IP
10. The JMP AX instruction executes a near jump to the offset address found in the AX register.
12. The JMP [DI] instruction is a near jump that loads IP with the offset address stored at the data segment location addressed by DI. The JMP FAR PTR [DI] instruction loads a 32-bit number from the data segment memory location addressed by DI into CS and IP, to effect a far jump.
14. The JA (jump if above) instruction jumps if the result of the most recent instruction was above. For example, if AX contains a 12H and the CMP AX,10H instruction is executed, the result shows that AX (12H) is above 10H.
16. JE, JNE, JG, JGE, JL, and JLE

18. JA and JBE
20. SETZ AL
22. ECX
24.
```
        MOV     DI,OFFSET DATA
        MOV     CX,150H
        XOR     AL,AL
LOOPS:
        MOV     ES:[DI],AL
        INC     DI
        LOOP    LOOPS
```
26. A procedure is a group of instructions stored in the memory one time, which can be used many times with the CALL instruction.
28. The near RET instruction removes a 16-bit number from the stack and places it into the IP register.
30. PROC
32. The RET 6 instruction adds a 6 to the SP register before returning. This instruction is often used to delete data stored on the stack within a procedure.
34.
```
MULTS  PROC    NEAR

       PUSH    BX
       PUSH    DX
       MOV     AX,DI
       MUL     SI
       MOV     BX,100H
       DIV     BX
       POP     DX
       POP     BX
       RET

MULTS  ENDP
```
36. An interrupt is a hardware- or software-initiated call to a procedure that interrupts the current software.
38. 256
40. An interrupt vector contains the address of the interrupt service procedure. The first two bytes contain the offset address and the last two bytes contain the segment address.
42. The IRETD instruction is used to return from an interrupt procedure where the contents of CS and EIP have been placed on the stack.
44. 40H × 4, or 100H through 103H.
46. INT 17H
48. WAIT
50. 16
52. ESC or escape

## CHAPTER 7

2. TEST.OBJ, TEST.LST, and TEST.CRF
4. The execute file (.EXE) can be of any length and can contain any number of segments, while the command file (.COM) can only contain one segment and is limited in length to 64K bytes.

6. The EXTRN directive indicates that a label is external to the current programming module.
8. The library file contains a collection of procedures that are selectively loaded into the linked execute file by the linker program.
10. A macro sequence is a group of instructions assigned to a new opcode or directive.
12.
```
ADD32  MACRO
       ADD     AX,CX
       ADC     BX,DX
       ENDM
```
14. The LOCAL directive indicates the labels that are local to the macro. Local causes these labels to be assigned different values when the program assembles.
16.
```
ADDM   MACRO LIST,LENGTH
       LOCAL ADDM1,ADDM2
       XOR AX,AX
       MOV SI,OFFSET LIST
       MOV CX,LENGTH
ADDM1:
       ADD AL,[SI]
       INC SI
       JNC ADDM2
       INC AH
ADDM2:
       LOOP ADDM1
       ENDM
```
18.
```
RANDOM   PROC NEAR

         INC   CL
         MOV   AH,6
         MOV   DL,0FFH
         INT   21H
         JZ    RANDOM
         RET

RANDOM   ENDP
```
20.
```
STRING   PROC    NEAR

         MOV     AH,6
         MOV     SI,DX
STRING1:
         LODSB
         CMP     AL,0
         JE      STRING2
         MOV     DL,AL
         INT     21H
         JMP     STRING1
STRING2:
         RET

STRING   ENDP
```
22.
```
MOVE     PROC NEAR

         MOV   AH,2
         MOV   DX,0306H
         MOV   BH,0
         INT   10H
         RET

MOVE           ENDP
```

24. By dividing it by ten and saving the remainder after each division
26. 30H
28. Each digit is decreased in value by 30H to convert to BCD. Then the result is multiplied by ten and the BCD results are added one digit at a time.
30.
```
LOOK   PROC NEAR

       MOV  BX,OFFSET CS:TAB1
       XLAT
       RET

LOOK   ENDP

TAB1:
       DB   30H,31H,32H,33H
       DB   34H,35H,36H,37H
       DB   38H,39H,41H,42H
       DB   43H,44H,45H,46H
```
32. XLAT SS:LOOK
34.
```
CODE     SEGMENT   'code'
         ASSUME    CS:CODE
MAIN     PROC   FAR
         MOV    AX,CS
         MOV    DS,AX
         MOV    CX,8
         MOV    SI,8000H
MAIN1:
         ROL    SI,1
         MOV    DX,OFFSET MES1
         MOV    AH,9
         INT    21H
         MOV    AX,8
         SUB    AX,CX
         ADD    AL,30H
         MOV    DL,AL
         MOV    AH,6
         INT    21H
         MOV    DX,OFFSET MES2
         MOV    AH,9
         INT    21H
         CALL   DISP
         LOOP   MAIN1
         MOV    AX,4C00H
         INT    21H
MAIN     ENDP
MES1     DB     13,10,' $'
MES2     DB     13,10,'2 = $
DISP     PROC   NEAR
         MOV    DI,0
         MOV    BX,SI
         CMP    BX,100
         JB     DISP1
         MOV    DL,'1'
         MOV    AH,6
         INT    21H
         SUB    BX,100
         MOV    DI,1
DISP1:
         MOV    AX,BX
         AAM
         CMP    AH,0
```

```
            JNE     DISP2
            CMP     DI,1
            JNE     DISP2
            PUSH    AX
            MOV     DL,AH
            ADD     DL,30H
            MOV     AH,6
            INT     21H
            POP     AX
DISP2:
            MOV     AH,6
            MOV     DL,AL
            ADD     30H
            INT     21H
            RET
DISP        ENDP
CODE        ENDS
            END     MAIN
```

36.
```
.286
CODE        SEGMENT    'code'
            ASSUME     CS:CODE
MES1        DB         13,10,'Segment start address = $'
MAIN        PROC    FAR
            MOV     AX,CS
            MOV     DS,AX
            MOV     AH,9
            MOV     DX,OFFSET MES1
            INT     21H
            CALL    GETADR
            MOV     DS,BX
            MOV     SI,0
            CALL    CRLF
MAIN1:
            CALL    DISPA
            CALL    SPACE
MAIN2:
            LODSB
            CALL    DISPB
            CALL    SPACE
            MOV     AX,SI
            AND     AX,000FH
            JNE     MAIN2
            CALL    CRLF
            MOV     AX,SI
            AND     AX,00FFH
            JNE     MAIN1
            MOV     AX,4C00H
            INT     21H
MAIN        ENDP
GETADR      PROC    NEAR
            XOR     BX,BX
            MOV     CX,4
GETADR1:
            CALL    GETCHAR
            SHL     BX,4
            ADD     BL,AL
            LOOP    GETADR1
            CALL    CRLF
            CALL    CRLF
            RET
GETADR      ENDP
```

```
GETCHAR  PROC   NEAR
         MOV    AH,6
         MOV    DL,0FFH
         INT    21H
         JE     GETCHAR
         CMP    AL,'0'
         JB     GETCHAR
         CMP    AL,'9'
         JBE    GETCHAR1
         CMP    AL,'a'
         JBE    GETCHAR1
         CMP    AL,'f'
         JA     GETCHAR
         SUB    AL,20H
GETCHAR1:
         CMP    AL,'F'
         JA     GETCHAR
         MOV    DL,AL
         INT    21H
         SUB    AL,30H
         CMP    AL,10
         JB     GETCHAR2
         SUB    AL,7
GETCHAR2:
         RET
GETCHAR  ENDP
CRLF     PROC   NEAR
         MOV    AH,9
         MOV    DL,13
         INT    21H
         MOV    DL,10
         INT    21H
         RET
CRLF     ENDP
SPACE    PROC   NEAR
         MOV    AH,6
         MOV    DL,20H
         INT    21H
         RET
SPACE    ENDP
DISPA    PROC   NEAR
         MOV    CX,4
DISPA1:
         MOV    AX,SI
         XCHG   AH,AL
         CALL   DISPB
         MOV    AX,SI
         CALL   DISPB
         MOV    AH,6
         MOV    DL,':'
         INT    21H
         RET
DISPA    ENDP
DISPB    PROC   NEAR
         PUSH   AX
         SHR    AL,4
         CALL   DISPC
         POP    AX
         CALL   DISPC
         RET
DISPB    ENDP
```

```
DISPC    PROC   NEAR
         AND    AL,15
         ADD    AL,30H
         CMP    AL,39H
         JBE    DISPC1
         ADD    AL,7
DISPC1:
         MOV    DL,AL
         MOV    AH,6
         INT    21H
         RET
DISPC    ENDP
CODE     ENDS
         END    MAIN
```

38. 64000
40. The bit mask register selects bits to modify.
42. 
```
CODE     SEGMENT   'code'
         ASSUME    CS:CODE
MAIN     PROC   FAR
         MOV    AH,0
         MOV    AL,13H
         INT    10H
         MOV    AX,0A000H
         MOV    ES,AX
         MOV    DI,0
         MOV    CX,200
         MOV    AX,0202H
MAIN1:
         STOSW
         DEC    DI
         LOOP   MAIN1
MAIN     ENDP
CODE     ENDS
         END    MAIN
```
48. Mini-floppy disk
50. 360K bytes
52. 1.44M bytes
54. The FAT contains information that indicates whether a cluster is free, bad, or occupied. If a cluster is occupied, the FAT contains the address of the next cluster in a file, or an FFFFH to indicate the last cluster.
56. 128
58. 8 plus 3 for an extension
60. 32
62. 
```
MOV   AH,3CH
MOV   CX,2
MOV   DX,OFFSET FILES
INT   21H
```
64. 
```
READ   MACRO   BUFFER,COUNT,HANDLE
       MOV   AH,3FH
       MOV   BX,HANDLE
       MOV   CX,COUNT
       MOV   DX,OFFSET BUFFER
       INT   21H
       ENDM
```
66. 
```
DATA   SEGMENT
FILEN  DB        'RAN.FIL',0
BUF    DB        100 DUP(?)
```

```
DATA   ENDS
CODE   SEGMENT   'code'
       ASSUME    CS:CODE,DS:DATA
MAKE   PROC FAR
       MOV  AX,DATA
       MOV  DS,AX
       MOV  AH,3CH
       XOR  CX,CX
       MOV  DX,OFFSET FILEN
       INT  21H
       MOV  BX,AX
       MOV  SI,200
MAKE1:
       MOV  AH,40H
       MOV  CX,100
       MOV  DX,OFFSET BUF
       INT  21H
       DEC  SI
       JNZ  MAKE1
       MOV  AH,3EH
       INT  21H
       MOV  AH,4C00H
       INT  21H
MAKE   ENDP
CODE   ENDS
       END  MAKE
```

# CHAPTER 8

2. Word integers (16 bits) range in value from –32,768 to +32,767; short integers (32 bits) range in value from $\pm 2 \times 10^{+9}$; and long integers (64 bits) range in value from $\pm 9 \times 10^{+18}$.
4. Single-precision (32 bits), double-precision (64 bits), and extended-precision (80 bits)
6. (a) –7.5, (b) +0.5625, (c) +282, (d) +2.0, (e) +10, and (f) +0.0
8. The microprocessor is usually free to execute integer instructions, while the coprocessor executes its own instructions.
10. The IE bit indicates an invalid error which can be a stack overflow, stack underflow, negative result of the FSQRT instruction, and so forth.
12. The rounding control bits of the control register select the rounding mode.
14. 00FCH–00FFH
16. NAN (not-a-number) is an invalid floating-point result.
18. Truncate toward zero
20. ESCape
22. (a) FROG DQ 23.44
    (b) DATA3 DD –132
    (c) DATA1 DD –23.8
    (d) DATA2 DQ ?
24. Loads the integer stored at memory location DATA to the top of the coprocessor stack.
26. The FADD instruction adds the data at ST(1) to ST, and pops ST(1) from the stack while replacing ST with the sum.
28. The FBSTP DATA instruction pops the top of the stack and stores its BCD equivalent in memory location DATA.

30. The FTST instruction compares ST with zero and the FXAM instruction examines ST.
32. IE
34. FLD1
36. FSTENV
38.
```
AREA    PROC  NEAR

        FLD   L
        FMUL  W
        FSTP  A
        RET

AREA    ENDP
```
40.
```
ROOTS   PROC  NEAR
        FLD1
        FSTP  CS:TEMP
        MOV   SI,OFFSET CS:ROOTS
        MOV   CX,9
LOOPS:
        FLD1
        FADD  CS:TEMP
        FST   CS:TEMP
        FSQRT
        FSTP  DWORD PTR CS:[SI]
        ADD   SI,4
        LOOP  LOOPS
        RET

TEMP    DD    ?
ROOTS   DD    9 DUP (?)

ROOTS   ENDP
```
42. The FNSTSW instruction does not use the FWAIT.
44.
```
SUM     PROC  NEAR

        MOV   ESI,OFFEST ARRAY1-8
        MOV   EDI,OFFSET ARRAY2-8
        MOV   EBX,OFFSET ARRAY3-8
        MOV   ECX,100
LOOPS:
        FLD   QWORD PTR [ESI+8*ECX]
        FMUL  QWORD PTR [EDI+8*ECX]
        FSTP  QWORD PTR [EBX+8*ECX]
        LOOPD LOOPS
        RET

SUM     ENDP
```

## CHAPTER 9

2. 64T bytes
6. The 80386 memory system is divided into four 8-bit-wide sections or banks. The four banks may be selected singly, in pairs, or all together to read/write bytes, words, and doublewords of data. The $\overline{BE0}$–$\overline{BE3}$ (bank selection signals) enable memory banks for data transfers.

8. Pipelining lengthens access time because, as the microprocessor uses data from one location, the memory system can access a second location.
10. 0000H through FFFFH
12. Earlier Intel microprocessors contained a $\overline{RD}$ and $\overline{WR}$ connection, while the 80386 contains a $\overline{R}$/W connection.
14. The $\overline{BS16}$ allows the 80386DX to function in a system that contains a 16-bit data bus.
16. EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP, and EFLAGS
18. CR0 is the machine control register that selects how the 80386 operates in a system: CR1 is not used, CR2 is the page fault address used with the debug or trap interrupt, and CR3 contains the page directory base address for memory paging.
20. Level 2
22. The bit scan reverse instruction scans a number from its left-most bit toward its right-most bit. If a one-bit is encountered, the zero flag bit is set and bit position of the one-bit is placed into the destination register.
24. MOV EAX,FS:[DI]
26. No
28. Interrupt type 7 is used to emulate the coprocessor if one does not exist.
30. A double interrupt fault occurs whenever a type 14 interrupt occurs with a type 10, 11, 12, or 13 interrupt.
32. A descriptor describes the starting location, length, and access rights for a protected-mode segment.
34. The selector addresses an entry in the global or local descriptor table that contains the descriptor that describes the selected memory segment.
36. 8,192
38. The segment descriptor describes a memory segment, while the system descriptor describes the properties of an interrupt, procedure, task, or jump address.
40. Via the task register
42. The PE bit is set in CR0.
44. The base address of the page directory is loaded into CR3.
48. The $\overline{FLUSH}$ is used to erase or invalidate the cache in the 80486 microprocessor.
50. The flags are basically the same, except for an alignment check flag that is added to the 80486 microprocessor EFLAG register.
52. Even
54. 16
56. Data are written to both memory and the cache in a write-through cache.
58. Yes, by changing a bit in CR0

## CHAPTER 10

2. (a) 256, (b) 2,048, (c) 4,096, and (d) 8,192
4. $\overline{CS}$ or $\overline{CE}$ selects the memory component for a read or a write operation.
6. The $\overline{WE}$ causes the RAM to write data to memory if its $\overline{CS}$ input is also active.
8. Too slow
10. The $\overline{G}$ pin enables reading, the $\overline{W}$ pin enables writing, and the $\overline{S}$ pin selects the RAM.

12. Dynamic RAM
14. The $\overline{CAS}$ selects the DRAM and also strobes half the address into an internal column latch. The $\overline{RAS}$ pin strobes half the address into a row address latch and also enables the refresh amplifiers.
16. The memory device must be selected to function at a particular range of memory locations.
18. See Figure D–1.
20. See Figure D–2.
22. The 74LS139 decoder is a dual 2-to-4 line decoder.
26. They select a memory read and memory write operation.
28. See Figure D–3.
32. 5
34. 1
36. Used to select the upper and lower memory banks
38. Upper
40. The microprocessor only samples the part of the bus that it needs to perform an operation.
46. Yes, if the data bus width is increased to 128 bits
48. Yes
50. CA and RA pins
56. Byte, set, and tag

**FIGURE D–1**

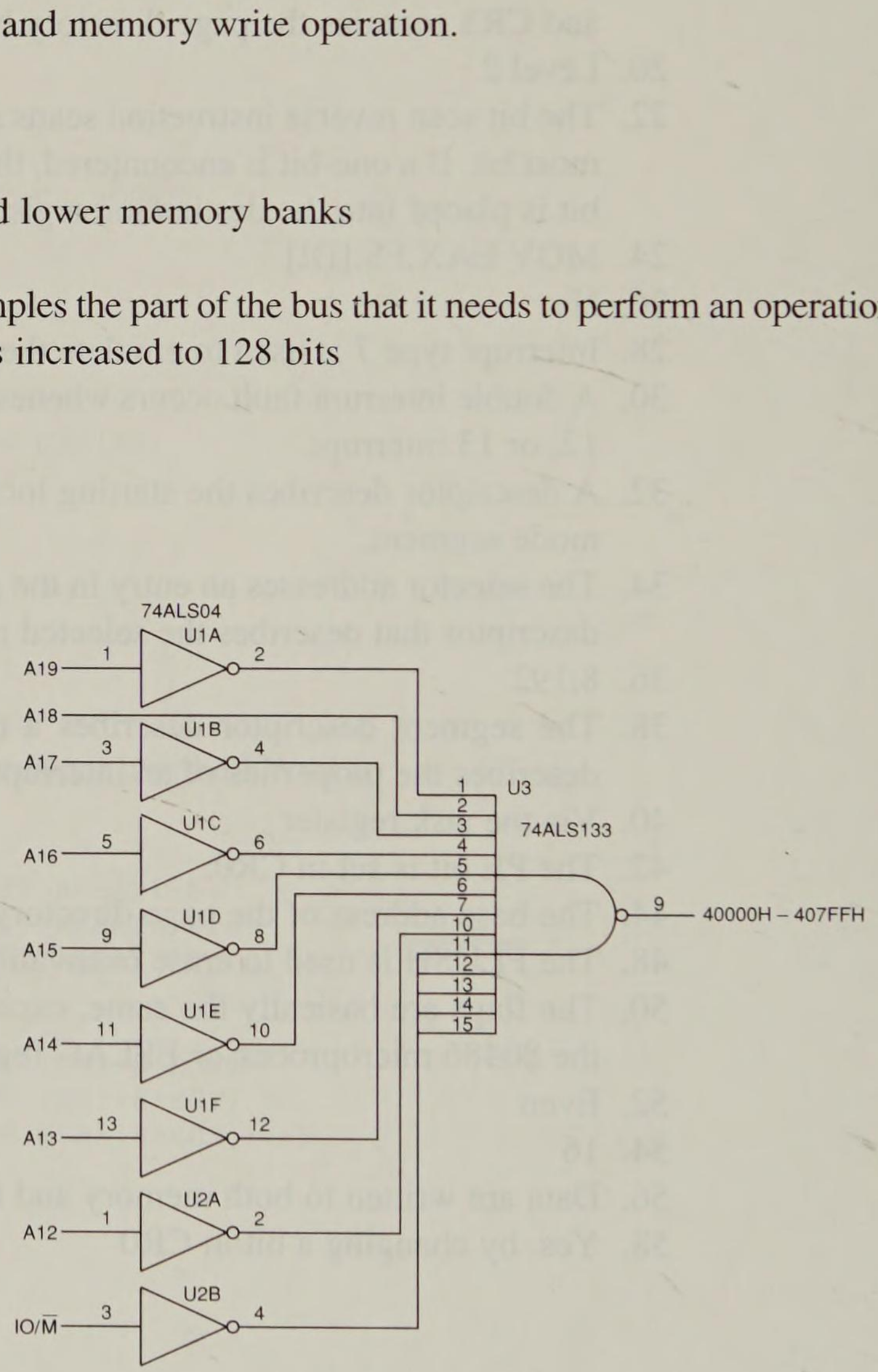



**FIGURE D–2**

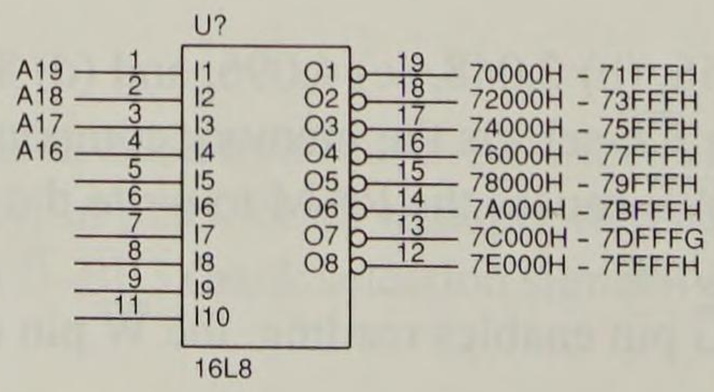


**FIGURE D–3**

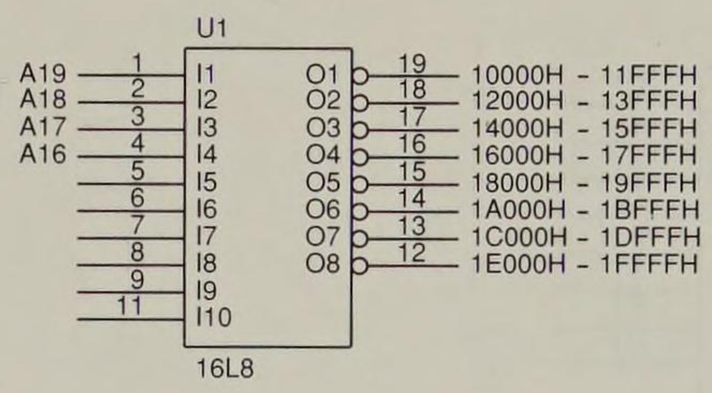



**FIGURE D–4**

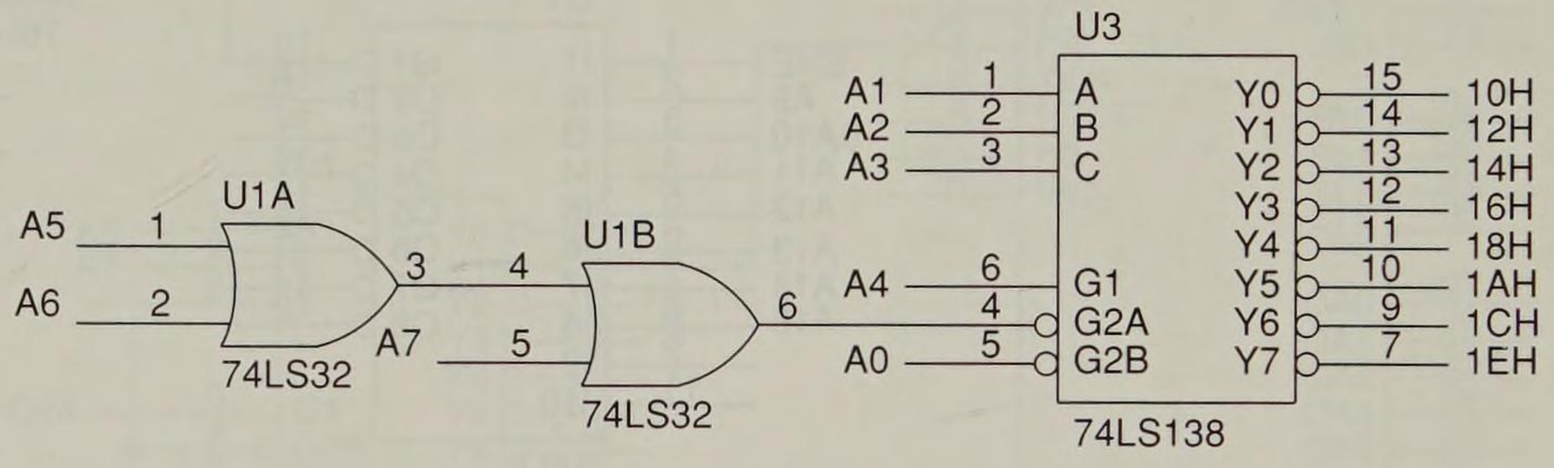
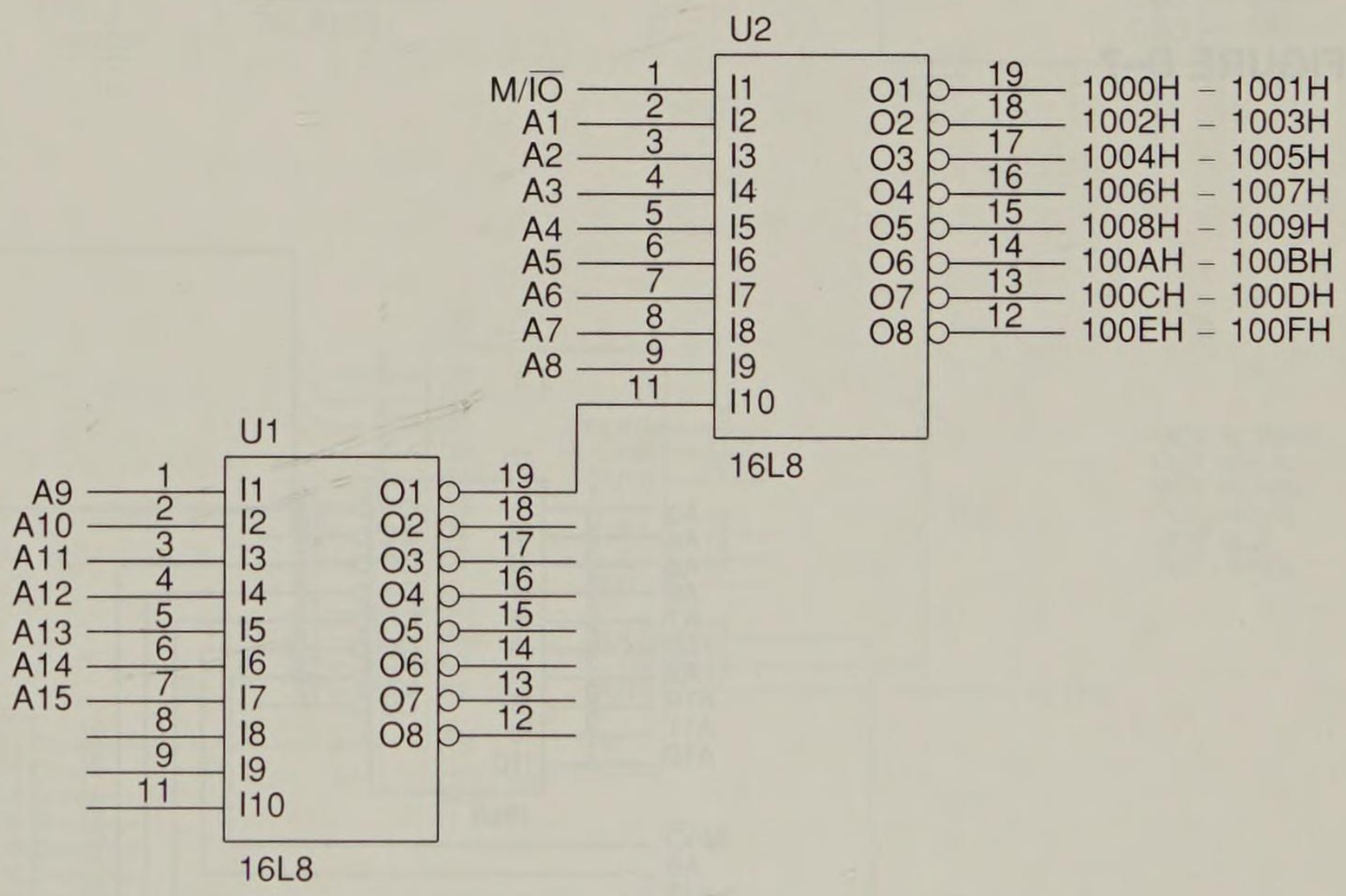



**FIGURE D–5**




## CHAPTER 11

2. In the byte following the opcode
4. DX
6. The OUTSB copies the byte contents of the data segment memory location addressed by SI into the I/O port addressed by DX. After the output, the contents of SI either decrement or increment by one depending on the condition of the direction flag.
8. The memory-mapped system uses a portion of the memory for I/O and the isolated I/O system uses a separate I/O space for I/O operations.
10. The basic output interface is a latch that captures output data.
12. Low
14. See Figure D–4.
16. See Figure D–5.

**FIGURE D–6**

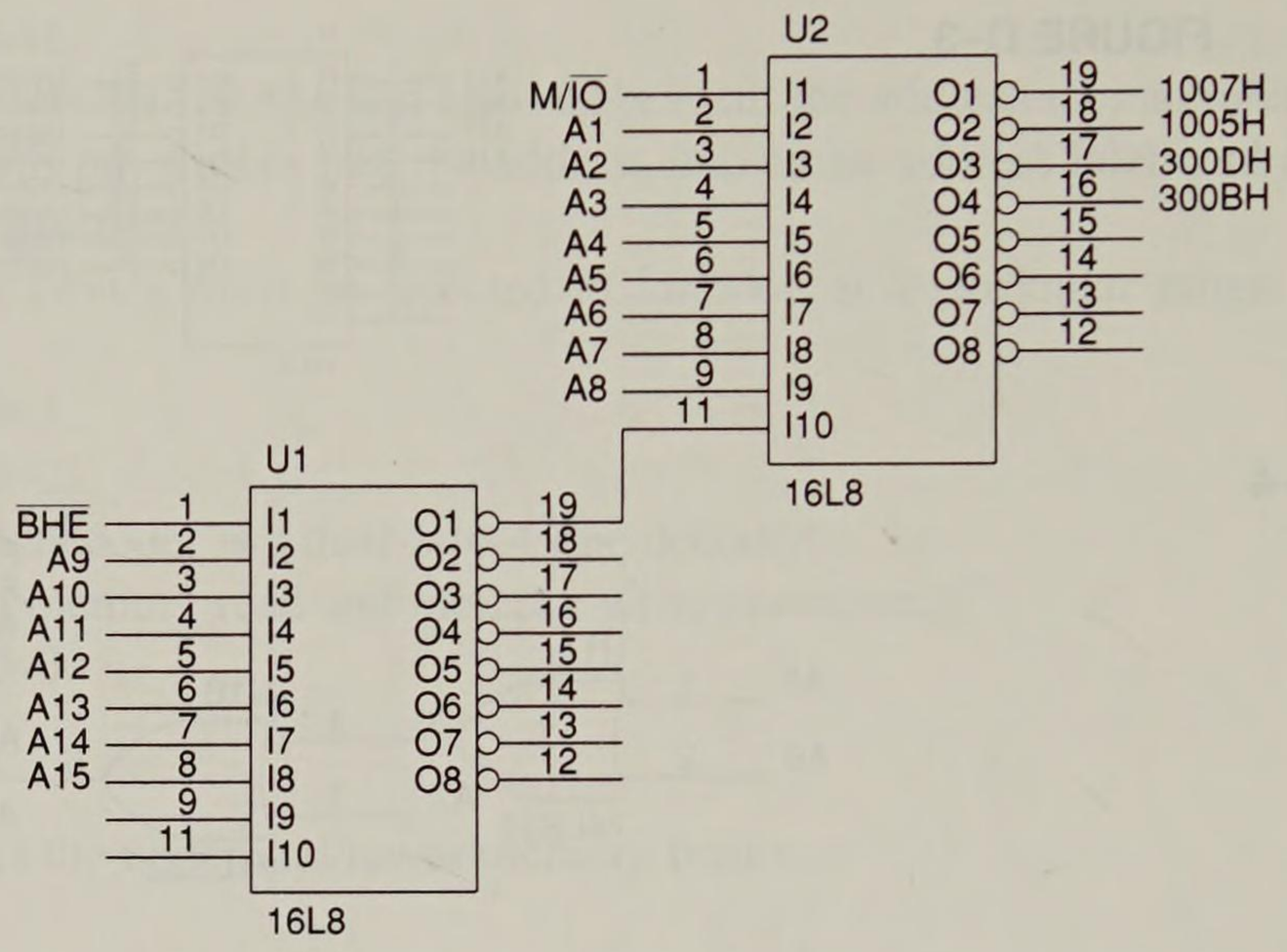
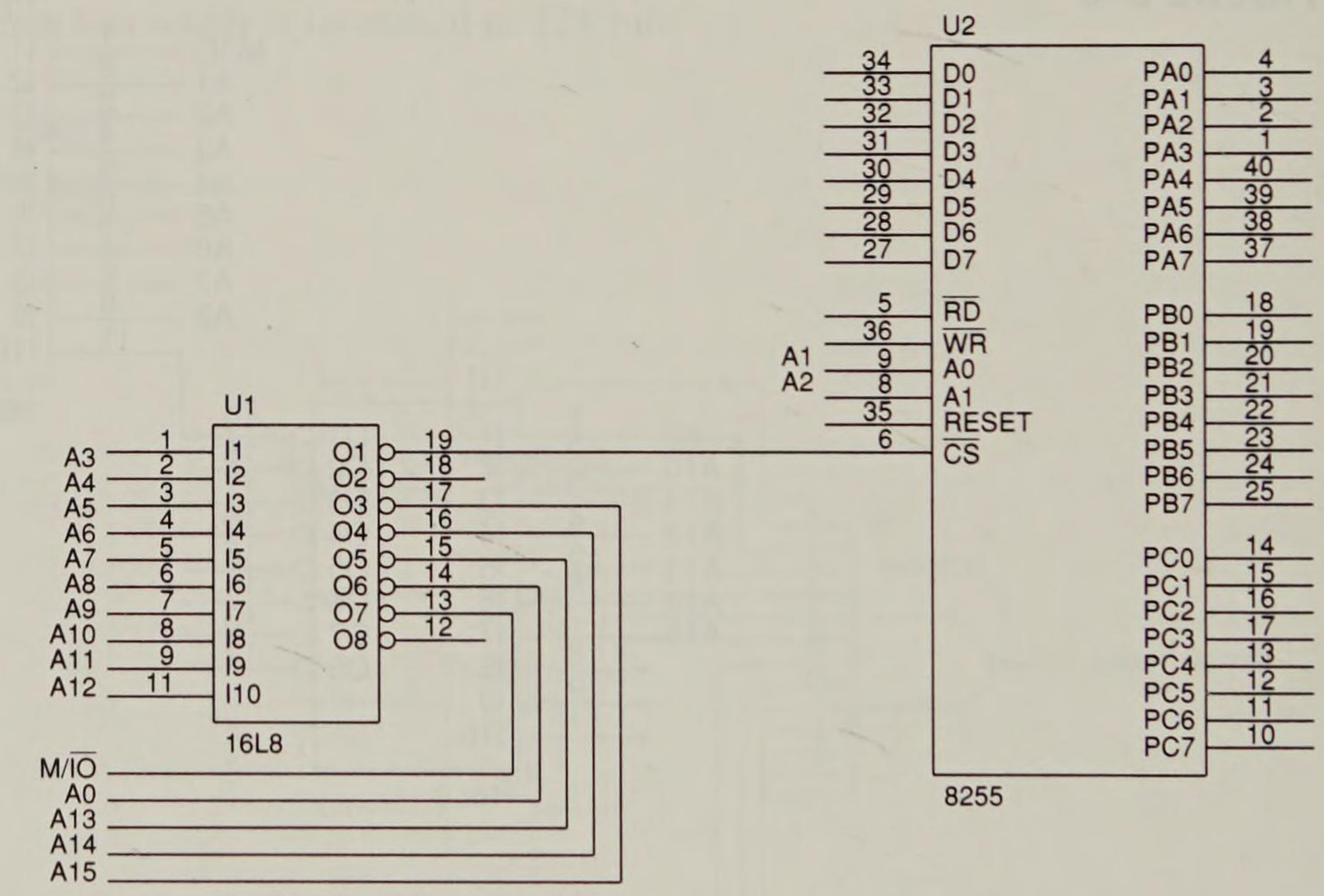
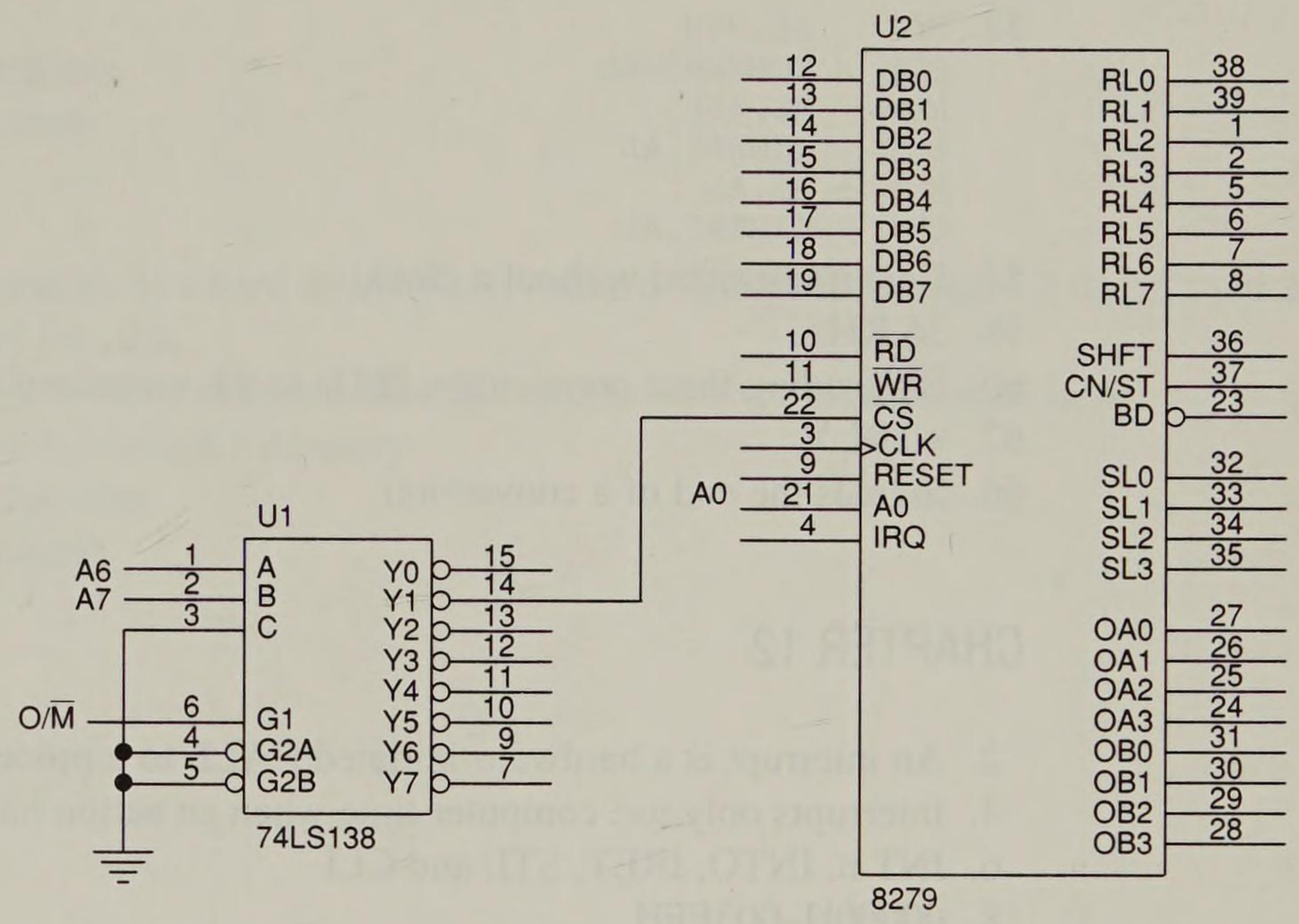



**FIGURE D–7**




18. See Figure D–6.
20. D0–D7
22. 24
24. A0 and A1
26. See Figure D–7.
28. Latched I/O, strobed I/O, and bidirectional I/O
30. The armature rotates each time current passes through a coil.
32.
```
MOV    AL,15
OUT    COMMAND,AL
```
34. It clears the $\overline{\text{OBF}}$ signal.
36.
```
IN     AL,PORTC
TEST   AL,10H
JNZ    PC4_IS_ONE
```

**FIGURE D–8**



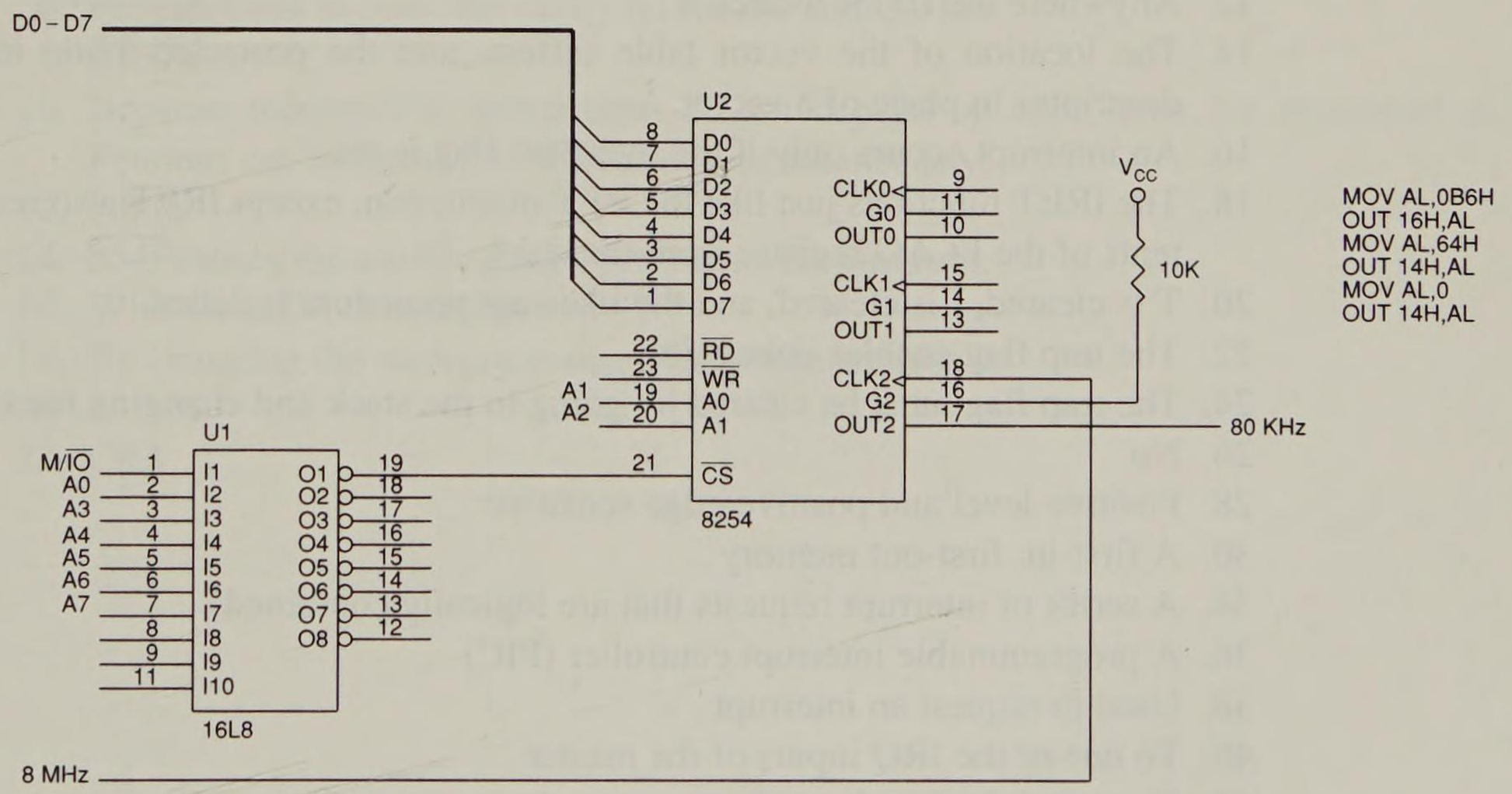




**FIGURE D–9**

38. PC0–PC2
40. 2
42. An overrun occurs when the data in the receiver is not removed before the next data are received.
44. See Figure D–8.
46. 10 MHz
48. See Figure D–9.
50. Least significant byte

52.
```
MOV     AL,74H
OUT     COMMAND,AL
MOV     AL,65H
OUT     TIMER1,AL
XOR     AL,AL
OUT     TIMER1,AL
```

54. Data transmitted without a clock
58. 36 KHz
60. By sending three consecutive 00Hs to the command register, followed by a 40H
62. +0.01 V
66. Signals the end of a conversion

## CHAPTER 12

2. An interrupt is a hardware-initiated CALL to a procedure.
4. Interrupts only use computer time when an action has occurred.
6. INT n. INTO, IRET, STI, and CLI
8. 00000H–003FFH
10. 00H–1FH
12. Anywhere the IDTR locates it
14. The location of the vector table differs, and the protected-mode interrupt uses a descriptor in place of a vector.
16. An interrupt occurs only if the overflow flag is set.
18. The IRET functions just like the RET instruction, except IRET also removes the contents of the FLAG register from the stack.
20. T is cleared, I is cleared, and the interrupt procedure is called.
22. The trap flag enables debugging.
24. The trap flag must be cleared by going to the stack and changing the trap flag state.
26. No
28. Positive-level and positive-edge sensitive
30. A first-in, first-out memory
34. A series of interrupt requests that are logically combined
36. A programmable interrupt controller (PIC)
38. Used to request an interrupt
40. To one of the IRQ inputs of the master
42. Operational control word
44. ICW2

## CHAPTER 13

2. The microprocessor stops executing software and floats its address, data, and control buses.
4. An I/O device to memory
6. $\overline{\text{DACK}}$
8. The microprocessor is idle and the DMA controller governs the system.

10. 4
12. Command register
16. Mini-floppy disk
18. Tracks
20. Cylinder
24. The head assembly in a hard disk drive is called a flying head because it doesn't touch the surface of the disk.
26. The voice coil is more accurate than the stepper motor and also much faster.
28. A compact disk read-only memory
30. Red, green, and blue
32. A picture element
36. 2M colors
38. 540

## CHAPTER 14

2. 64 bits
4. 66 MHz
6. Provides and accepts the parity bit for the address bus
8. $\overline{\text{BRDY}}$
10. Because independent instructions do not depend on each other for execution, the Pentium can execute two instructions simultaneously.
12. 5.8 ns
14. $\overline{\text{SMI}}$ causes the memory management mode interrupt.
16. With the RSM instruction
18. By changing the memory management base address
20. 1
22. CR4

# INDEX


AAA, 155
AAD, 156
AAM, 156
AAS, 157
Abacus, 2
Access:
  rights, 52, 342
  time, 383
Accumulator, 39–40
ADC, 141
ADD, 137
Addition, 137–143
  with carry, 141–143
Address:
  bus, 22
  column, 389
  connections, 380–381
  decoding, 391–400
  offset, 44
  row, 389
  segment, 44
Addressing modes:
  base-plus-index, 71–73
  base-relative-plus-index, 75–77
  coprocessor, 286
  direct, 65–67
  immediate, 64–65
  register, 63–64
  register indirect, 67–70
  register relative, 73–75
  scaled-index, 77–78, 97–99
  special, machine language, 97
$\overline{\text{ADS}}$, 336
ALIGN, 124
Alignment flag bit, 43
Altair 8800, 5
Analog-to-digital (ADC) interface, 508–511
Analytical Engine, 2
AND, 157–159
ASCII, 24–27, 65, 124, 155–157
  displaying to video, 216–217
  extended keyboard data, 213
  to binary conversion, 225–226
ASCII-Z string, 252
Assembler, 26, 101, 120–131, 204
Attributes:
  file, 253
  video, 221
AUTOEXEC.BAT, 20
Auxiliary flag bit. *See* Carry

Babbage, Charles, 2
Bank, 410, 416, 623
Base
  address, 51, 342
  index register, 40
  plus-index addressing, 71–73
  pointer register, 41
  relative-plus-index addressing, 75–77
Battery-backed-up memory, 532
BCD (Binary-coded decimal), 4, 27, 154–155, 275, 554
  to 7-segment conversion, 228–229
Big endian, 29

Binary-to-ASCII conversion, 223–225
BIOS, 12, 14
  video, 218–221
Bit, 4
  scan instructions, 167–168
  test, 162–163
Bit-mapped display, 235
Bit mask register, 239
$\overline{\text{BLE}}$ (Bus low enable), 409
$\overline{\text{BHE}}$ (Bus high enable), 409
BIST, 630
Boot sector, 248
Borrow, 144–146
BOUND, 197
Branch prediction, 627–628
BSF, 167–168
BSR, 167–168
BSWAP, 120
BT, BTC, BTR, and BTS, 163
Bus, 10, 22–23
  arbiter, 580–583
  local, 577
  master, 578
  remote, 577
$\overline{\text{BUSY}}$, 196
Byte, 4, 21, 28–29
  registers, 39

Cache, 330, 371, 437–439, 628
CALL, 186–189
Carry:
  auxiliary flag bit, 42
  control instructions, 196
  flag bit, 41–42, 141, 144, 148, 164, 196
CBW, 151
CD-ROM, 606–607
CDQ, 153
Centronics parallel interface, 451
CGA, 609
Chip selection, memory, 381
CISC, 6
CLC, 196
CLD, 109–110
CLI, 194, 526
Cluster, 249, 604
CMC, 196
CMP, 146
CMPS, 169–170
CMPXCHG, 147
Code segment register, 43–44
Color, video, 222
Colossus, 2
Column address strobe ($\overline{\text{CAS}}$), 389
Command file (.COM), 205
COMMAND.COM, 20
Communications interface, 496–506
Comparison, 146–147
Complement, 163
Computer system block diagram, 10
CONFIG.SYS, 18–20
Control:
  bus, 23
  registers, 338–339
  signals, 333
Coprocessor, numeric:
  architecture, 278–283
Count register, 40
CPU, 21
Cursor, 218
CWD, 152
Cylinder, 249, 599

DAA, 154
DAS, 155
Data:
  addressing modes, 60–81
  ASCII, 24–27, 124
  BCD, 27
  bus, 22
  byte, 28–29, 121–124
  constant, 64
  doubleword, 30–31, 121–124
  floating-point, 31–33, 275–277
  immediate, 64–65, 99–100
  integer, signed, 247–248
  register, 41
  segment register, 44, 100–101
  structures, 78–81
  variable, 64
  word, 29–30, 121–124
DB, 26, 27, 29, 121
DD, 31, 121
Debug register, 339–340
DEC, 144
Decoder:
  74LS138, 393–396
  I/O, 452–459
  NAND gate, 391–393

PLD, 398–400
PROM, 396–397
Decrement, 144
Descriptor, 50–54, 341
Destination index register, 41
Digital-to-analog converter, 506–508, 610
Direct addressing, 65–67
Direct memory access. *See* DMA
Direction flag bit, 43, 109–110
Directives, 121–126
Directory:
disk, 250
page, 357–358
Displacement, 45, 94
Display interface, 483–485
DIV, 152
Division, 150–154
DMA (Direct memory access):
basics, 560–561
controller, 562–577
memory-to-memory transfer, 569–573
printer interface, 575–577
Double-clocked, 8
Doubleword, 21, 30–31
registers, 39
DOS, 12, 18–21
DQ, 121
DRAM, 388–391, 427–437
DSDD (double-sided, double-density), 600, 602
DT, 121
DUP, 121–124
Duplex, 497
DW, 29, 30, 121

EHD (Extended high-density), 602
EISA, 11
EMM386.EXE, 19
EMS, 15
END, 128
ENDP, 125–126, 186
ENDS, 123
ENTER, 197–199
EQU, 124–125
ESC, 197
ESDI (Enhanced small disk interface), 605
Exclusive-OR, 160–161
Execution file (.EXE), 204
Extra segment register, 44
EXTRN, 206–207

Far, 81, 179–180
FAT (File allocation table), 249–250
File, 247–267
attribute, 253
close, 254–255
creation, 251–253
handle, 253
open, 254–255
pointer, 255
random access, 259–261
read, 254–255
sequential access, 251
write, 253–254
Fixed disk. *See* hard disk
Flag register, 41–43, 138, 144, 181
Floating-point, 31–33, 275–277
Floppy disk, 248, 598–602
Flying head, 603

G, defined, 6
Granularity (G-bit), 51
GUI, 6

Handle, file, 253
Handshaking, 450–452
Hard disk, 603–604
HD (High-density), 600, 602
Head, 249
Hexadecimal, 12, 65
conversions, 226–228
HIMEM.SYS, 19
HLDA, 560
HLT, 196
Hoff, Marcian T.; 3
HOLD, 560
Hollerith, Herman, 3
Hook, 529

I/O (Input/Output):
address, 114, 118
decoder:
8-bit, 452–453, 455–458
16-bit, 453–458
32-bit, 458–459
instructions, 114–115, 117–118, 445–447
isolated, 446–447
memory-mapped, 447
privilege-level flag bit, 43
system, 16–17, 332–333, 447–448

IBM, 3
iCOMP, 9
IDE (Integrated drive electronics), 605–606
IDIV, 152
Immediate, 65–65, 99–100
IMUL, 149
IN, 117–118
INC, 140
INCLUDE, 212
Increment, 140–141
Index hole, 598
Indirect addressing, 67–70
Input interface, 449
INS, 114
Instruction:
  modes, 92–93
  pointer register, 41
INT, 192–193, 524
$\overline{\text{INTA}}$, 520
Interrupt, 191–195
  basic, 519–530
  control, 194
  daisy-chained, 539–540
  expansion, 537–540
  flag bit, 42–43
  instructions, 192–194
  keyboard, 533
  personal computer, 194–195
  procedure, 194
  programmable interrupt controller (PIC), 541–554
  vector, 12, 191–192, 520–523
Intersegment, 81
INTO, 194
INTR, 520
Intrasegment, 82
IRET, 193, 524
IRETD, 193, 524
ISA, 11

Jacquard's loom, 2
JMP, 81–83, 175
Jump, 81, 175–185
  conditional, 181–183
  far, 179–180
  near, 178–179
  short, 176–178

K, defined, 4
Keyboard:
  interface, 478–483
  read with echo, 213–214
  read without echo, 214–215
  real line, 215–216
Kilby, Jack, 3
KIP, 4

Label, 81
LAHF, 116
LEA, 107–108
LEAVE, 197–199
LDS, LES, LFS, LGS, and LSS, 108–109
Library files, 207–209
Limit, 51, 342
Linear:
  memory page, 357
  program, 210
Linker, 204–206
Little endian, 29
Loading, 327
LOCAL, 211
$\overline{\text{LOCK}}$, 197
LODS, 110–111
Lookup tables, 228–234
LOOP instructions, 184–185

M, defined, 5
Machine:
  language, 92
  status word (MSW), 338
Macro, 209–212
Map mask register, 239
Mask, 158
  bit mask register, 239
  map mask register, 239
Memory:
  absolute, 66
  address pins, 380–381
  cache, 437–439
  control pins, 381–382
  data pins, 381
  dynamic (DRAM), 388–391, 425–437
  error correction, 406–408
  expanded memory system (EMS), 15–16, 19
  extended memory system (XMS), 11, 19
  interface:
    8-bit, 400–404
    16-bit, 409–416
    32-bit, 416–420
    64-bit, 420–423
  interleaved, 423–427
  management, 340–349, 631
  maps of personal computer, 11, 13, 15

paging, 357–363
Pentium, 623
protected mode, 50–55
real mode, 44–50
selections pins, 381
static (SRAM), 385–388
symbolic, 66
system, 80386, 329
system area, 14–16
transient program area (TPA), 12–14
upper memory system, 15–16
MFM (Modified frequency modulation), 600, 605
Microprocessors:
data formats, 23–33
embedded, 6
history of, 4–9
introduction to, 21–23
memory size, 22
programming mode, 39–44
MIP, 6
MNI, 530–532
MOD, 94–95
Models, 126–129
Motor control, 492–496
MOV, 60, 92–101
MOVS, 113–114
MOVSX, 119–120, 152
MOVZX, 119–120, 151
MUL, 149
Multiplication, 148–150
Multitasking, 577

Near, 82, 178–179
Nested task flag bit, 43
Nibble, 4
NEG, 163
NOP, 196
NOT, 163
NRZ (nonreturn to zero), 600

OFFSET, 108
Offset, 44–50
Opcode, 60, 93–94
Optical disk memory. *See* CD-ROM
OR, 159–160
ORG, 125
OUT, 117–118
Output
enable, memory, 381
interface, 449–450
OUTS, 114–115
Overdrive, 8
Overflow flag bit, 43, 148

Paging, 357–363, 630
PAL, 398–400
Palette, 612
Paragraph, 45
Parity
checker/generator, 370–371, 404–406, 623
flag bit, 42
Pascal, Blaise, 2
PCI, 11–12
Physical page, 357
Pinouts:
2716, 2K × 8 EPROM, 383
44256, 389
62256, 32K × 8 SRAM, 388
74LS138, 393
74LS139, 396
74AS280, 404
80287, 278
80386DX, 326
80386SX, 326
80486, 364
8237A–5, 562
8251, 498
8254, 486
8255, 460
8259A, 541
8279, 476
8289, 581
ADC0804, 509
Centronics parallel interface, 451
DAC0830, 507
PAL16L8, 399
Pentium, 619
SIMM, 1M × 8, 435
TMS4016, 2K × 8 SRAM, 385
TMX4C1024, 1M × 1 DRAM, 390
Pipeline, 330
PLD, 398–400
Pointer, file, 255
POP, 85, 103–104
Port, 118
Power failure detection, 531
Prefix:
address size override, 95
LOCK, 197
register size override, 95
repeat, 112
segment override, 120

PEL, 235
Print spooler, 587–598
PROC, 124–126, 186
Procedure, 124–126, 185–191
  interrupt, 194
Program:
  coprocessor, 313–321
  invisible register, 54–55
  memory addressing modes, 81–83
  visible register, 39
Programmable peripheral interface (PPI), 459–475
Programming:
  8255, 416–475
  model, 39–44
  protected mode, 50–55
Pseudo-operations, 121
PTR, 69, 99–100, 121, 140, 144, 189
PUBLIC, 206–207
PUSH, 85, 102–103

R/M, 96–97
Raster, 612–614
RCL, 167
RCR, 167
READY, 336
Real
  mode, 44–50
  number. *See* Floating-point
Real-time clock, 554–556
Refresh, 388, 428
Register-addressing, 63–64, 67–70
Register-relative addressing, 73–75
Registers:
  80386, special, 338–340
  assignments, machine language, 95
  control, coprocessor, 282–283
  CS, 44
  default, 46–47
  DS, 44
  EAX, 39–40
  EBP, 41
  EBX, 40
  ECX, 40–41, 165, 169, 183
  EDI, 41, 110
  EDX, 41
  EFLAGS, 41–43
  EIP, 41
  ES, 44
  ESI, 41, 110
  ESP, 41
  extended, 39
  FS, 44
  GS, 44
  multipurpose, 39–41
  segment, 43–44
  special-purpose, 41–43
  SS, 44
  status, coprocessor, 279–280
  tag, coprocessor, 282–283
Relative jump, 176
Relocation, 47–50
Remainder, 153–154
REP, 112
REPE, 169
Repeat, 112, 168–169
REPNE, 169
Resume flag bit, 43
RET, 190–191
Return address, 185
RGB, 608–609
RISC, 9
RLL (Run-length limited), 604–605
ROL, 167
Root directory, 248
ROR, 167
Rotates, 166–167
Row address strobe ($\overline{\text{RAS}}$), 389

SAHF, 116
SAL, 164
SAR, 164
SBB, 144
Scaled-index addressing, 77–78
Scaling factor, 77, 97
SCAS, 168
SCSI (Small computer system interface), 605
Sector, 248, 598
Segment, 44–50
SEGMENT, 123
Selector, 50–54, 341
Set instructions, 183–184
Shift instructions, 164–166
SHL, 164
SHLD, 166
Short, 82, 176–178
SHR, 164
SHRD, 166
Sign flag bit, 42
SMM (System memory management), 631
Software, 21

Source index register, 41
SRAM, 385–388
Stack:
  coprocessor, 285
  initialization, 104–107
  memory addressing, 83–85, 102–107
  pointer register, 41, 84
  segment register, 44, 84
STC, 196
STD, 109–110
Stepper motor, 465–468
STI, 194, 526
Stored program concept, 21
STOS, 111–113
String instructions, 109–115, 168–170
STRUC, 78
SUB, 143
Subroutine. *See* Procedure
Subtraction, 143–146
  with borrow, 144–146
Superscaler, 628

Table, page, 358–363
Task state segment (TSS), 347–349
TEST, 161–163
$\overline{\text{TEST}}$, 196
Test register, 339–340
THIS, 124
Timer (8254), 485–496
Timing:
  80386, 333–337
  80486, 372–373
  Pentium, 625–627
TPA, 11, 12–14
Track, 248, 598
Trap flag bit, 42, 527
TTL video display, 608–609
Turing, Alan, 3
Two's complement form, 28

USE16 and USE32, 128
USES, 125

VESA, 12
Video display:
  16-color, 640 × 480 mode, 239–247
  256-color, 320 × 200 mode, 235–238
  analog RGB, 609–614
  BIOS, 218–221
  DAC, 610
  displaying character string, 216–217
  displaying single ASCII character, 216
  signals, 607–608
  text using direct memory access, 221–223
  TTL RGB, 608–609
  VGA, 234–247
Virtual mode
  8086, 356–357
  flag bit, 43

WAIT, 196, 425
  states, 335–336
Word, 21, 29–30
  registers, 39
WORM, 606
Write:
  enable ($\overline{\text{WE}}$), 382
  strobes, 410
WYSIWYG, 7

XADD, 143
XCHG, 116
XLAT, 116–117, 228
XMS, 11
XOR, 160–161

Yoke, 614

Zero flag bit, 42
Zilog, 5
Zuse, Konrad, 3

0-02-314260-X